„Virtuelle Realität und persistente Datenspeicherung mithilfe von XML“

von

Jürgen Diewald

Eine Diplomarbeit an der Fachhochschule Köln,

Abteilung Gummersbach - Fachbereich Informatik

Fachhochschule Köln
University of Applied Sciences Cologne

Campus Gummersbach

In Zusammenarbeit mit dem Fraunhofer Institut

für Medienkommunikation (IMK) in Sankt Augustin

Fraunhofer Institut Medienkommunikation

Betreuende Personen:

Prof. Dr. Heide Faeskorn-Woyke

Hr. Stefan Conrad

August 2004

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Virtuelle Realität ist ein interessantes Gebiet dessen Entwicklung schon seit vielen Jahren vorangetrieben wird. Die neuen Möglichkeiten die sich aufgrund technischer Fortschritte daraus ergeben, sind umwerfend und finden in vielen Bereichen Verwendung. Doch das Potential der VR ist noch viel größer.

Dazu passt es auch, das im Laufe der Jahre die Computer immer mehr Einzug in das Leben der Menschen nahmen. Gab es früher noch riesige Geräte, die mit Lochkarten arbeiteten und kaum aufwendige Aufgaben bewältigen konnten, so gibt es heute erheblich schnellere und kleinere Systeme, die zudem auch noch viel leistungsfähiger sind. Von den drastisch gesunkenen Kosten mal ganz abgesehen. Folglich findet auch die breite Öffentlichkeit Zugang zu leistungsfähigen Computersystemen, wodurch bei immer mehr Menschen das Interesse an Virtual Reality geweckt wird. Es ist ein enormer Markt und die Technologie hält immer wieder Einzug in neue Bereiche und ist aus manchen gar nicht mehr hinauszudenken.

Allerdings erfordert es auch ein entsprechendes Wissen, um leistungsfähige und zweckmäßige VR-Anwendungen erstellen zu können. Und so verwundert es nicht, das es, trotz der einfachen Verfügbarkeit, nicht viele Menschen gibt, die Anwendungen entwickeln können, welche wiederum die technologischen Fortschritte richtig ausnutzen. Bei der Entwicklung einer solchen Anwendung bedarf es sowohl künstlerischen Fähigkeiten, zur Modellierung von Gegenständen und der gesamten Szene, als auch technisches Wissen zur bestmöglichen Ausnutzung der Hardware und Software. Allerdings gibt es nur wenige Informatiker die zeitgleich auch künstlerisch befähigt sind und umgekehrt. [CNG02]

Hinzu kommt, das der technologische Fortschritt nicht halt macht. Durch ständige Weiterentwicklungen nimmt die Komplexität stetig zu und es ist angebracht, die Anwendungsentwicklung zu vereinfachen, so dass immer mehr Künstler die verfügbaren

Möglichkeiten auch eigenständig ausschöpfen können. Bei dieser Vereinfachung gilt es hauptsächlich, den Programmieraufwand zu verringern, sprich die erforderlichen Programmiersprachenkenntnisse so gering wie möglich zu halten. Hierbei können schon entsprechende Entwicklungswerkzeuge Abhilfe schaffen.

Noch besser wäre es, den künstlerischen Teil vom technischen Teil komplett zu trennen. Eine einfache Schnittstelle würde es den Künstlern erlauben, eine virtuelle Szene in ihrem Aufbau, ihren Eigenschaften und den Interaktionsmöglichkeiten abstrakt zu beschreiben. Die zugrundeliegende Software würde dann aus dieser abstrakten Szenenbeschreibung die virtuelle Welt generieren, ohne besondere Programmierkenntnisse der beteiligten Personen vorauszusetzen. Dies würde vermutlich vielen Menschen das Tor zur Entwicklung von VR-Anwendungen öffnen.

In diesem Zusammenhang ist es erst einmal nötig, das die verwendete VR-Software über ein System zur Unterstützung von Persistenz verfügt. D. h. die zuvor erwähnten abstrakten Daten zur Beschreibung der Szene müssen auch irgendwie von der Software eingelesen und verwendet werden können. Genauso muss im umgekehrten Fall die Möglichkeit zur Speicherung der Datenstruktur innerhalb der Software bestehen, so dass die Daten von externen Programmen verwendet bzw. manipuliert werden können. Aufgrund der Standardisierung und Mechanismen zur Transformation, was wiederum in einer hohen Kompatibilität resultiert, eignet sich hierbei XML hervorragend als Datenformat.

Allerdings sollten hierbei bereits vorhandene Schnittstellen innerhalb der Applikation bestmöglich genutzt werden und nur minimale, unbedingt benötigte Änderungen an der Software-Umgebung vorgenommen werden. Auch das letztendliche Datenformat sollte ohne größere Umstellungen verwendbar sein, was durchaus im Gegensatz zur Kompatibilität zu anderen Formaten steht.

Im nächsten Kapitel namens *Virtuelle Realität* erfolgt eine Einführung in eben Jene. Neben der nötigen Hardware, werden auch noch die verschiedenen Arten der VR als auch einige ihrer Anwendungsgebiete betrachtet.

Das Kapitel danach beschäftigt sich mit dem Thema Persistenz und einigen Ansätzen zur Realisierung selbiger.

Als nächstes folgt ein Kapitel über die Virtual Reality Modeling Language (VRML). Diese wird teilweise auch schon als Datenformat für dreidimensionale Welten verwendet und könnte folglich hier von Nutzen sein.

Anschließend erfolgt das Kapitel *eXtensible Markup Language*, in der genau diese Auszeichnungssprache vorgestellt wird. Die XML-Technologie vereint die Vorzüge von proprietären Formaten mit einer hohen Kompatibilität.

Das nächste Kapitel beinhaltet schließlich den praktischen Teil dieser Diplomarbeit und beschreibt die Implementierung einer Software-Schnittstelle, die es dem Virtual-Reality-Software-Framework Avango ermöglicht, die zugrundeliegende Datenstruktur des Szenengraphen zu sichern und wiederherzustellen.

Im letzten Kapitel wird die gesamte Arbeit noch einmal zusammengefasst und ein Ausblick auf mögliche Optimierungen und Entwicklungen gegeben.

Kapitel 2

Virtuelle Realität

"Virtual Reality is a new plane of reality that will thrill everybody."

Jaron Lanier [ST98]

In diesem Kapitel erfolgt nun ein Überblick über die Virtuelle Realität (VR) und die damit verbundenen Technologien. Neben einer einleitenden Begriffsdefinition, den Voraussetzungen zur Realisierung einer VR-Umgebung und den unterschiedlichen Arten der VR, enthält dieses Kapitel abschließend noch einen Überblick über einige der Anwendungsgebiete der VR.

2.1 Definition

Im Laufe der 90'er Jahre erfreuten sich die Begriffe „Virtuelle Realität" und „Cyberspace" eines enorm wachsenden Bekanntheitsgrades. Wobei der Ausdruck „Cyberspace" bereits 1984 von dem amerikanischen Science-Fiction-Autor William Gibson in seinem Roman *Neuromancer* geprägt wurde und oftmals als Synonym für Virtuelle Realität verwendet wird. Dieser Roman war, vor allem in den USA, der Grundstein für eine postmoderne Bewegung mit Anhängern aus allen Gesellschaftsschichten die sich selbst als „Cyberpunks" bezeichneten. [LS03] Der Begriff „Virtual Reality" hingegen wurde 1989 von Jaron Lanier, dem Gründer von VPL Research, Inc.[1], eingeführt.

Die Zahl derer, die an der virtuellen Realität interessiert sind, steigt seit Jahren stetig an. Aufgrund der unterschiedlichen Interessen jener Gruppe entstehen immer wieder neu erschlossene Anwendungsgebiete, neue Hardwaregeräte und Softwarelösungen in diesem Bereich. Durch diese ständigen Neu- und Weiterentwicklungen entwickelt der

[1] VPL steht für Virtual Programming Language. Die Firma wurde 1984 gegründet und war Vorreiter auf dem Gebiet der Virtual Reality.

Begriff „Virtuelle Realität“ eine gewisse Eigendynamik, die eine eindeutige, oder besser gesagt allgemeingültige, Definition erschwert.

Jedoch verbindet, zumindest im klassischen Sinne, ein Punkt die verschiedenen Anwendungsgebiete. Und zwar die Immersion in eine künstliche Umgebung. Diese Umgebung wiederum wird von einem Verbund aus verschiedensten Soft- und Hardware-Komponenten erzeugt. Dabei wird versucht, natürliche Aspekte der menschlichen Wahrnehmung bestmöglich zu berücksichtigen und die Simulation dementsprechend zu erstellen, wie z. B. die Darstellung von visuellen Informationen in drei räumlichen Ebenen. Allerdings gibt es inzwischen auch Technologien, bei denen die Immersion nur noch eine unbedeutende Rolle spielt, bzw. gar nicht mehr eintritt.

Mit Hilfe von speziellen Eingabegeräten kann der Benutzer dann mit dieser künstlichen Welt interagieren. Somit ist dies die modernste Schnittstelle zwischen Mensch und Computer.

2.2 Technische Voraussetzungen

Die Möglichkeiten der VR-Technik sind von vielen Faktoren abhängig, unter anderem dem Entwicklungsstand passender Eingabe- und Ausgabegeräte, aber auch dem technischen Fortschritt in Teilgebieten der Informatik. Zum Beispiel im Bereich Netzwerke, Parallelrechner-Architekturen und künstliche Intelligenz. Natürlich spielt auch die Entwicklung entsprechender Software eine große Rolle.

Um ein VR-System zu realisieren benötigt man im allgemeinen einen Computer, entsprechende Software, sowie Geräte zur Ein- und Ausgabe. Abhängig von der verwendeten Software kontrolliert der Computer die angeschlossenen Geräte, berechnet die virtuelle Welt und erzeugt sie mithilfe der Ausgabegeräte.

Die obigen Komponenten wiederum sind abhängig vom jeweiligen Anwendungsgebiet. So liegen z. B. bei militärischen Anwendungen für Piloten die Prioritäten bei der Reaktionszeit des Systems, der Authentizität der Flugsimulation und der fotorealistischen Darstellung des Zielobjektes, während bei der Erschließung eines Massenmarktes von

VR-Anwendungen doch wohl eher die Kosten und Integrationsmöglichkeiten der VR-Ausrüstung in Standard-PC-Umgebungen im Vordergrund stehen dürften. [LS03]

2.2.1 Eingabegeräte

Sie sind nötig, um dem Benutzer jegliche Interaktion mit der virtuellen Welt zu ermöglichen. So halten sie z. B. die aktuelle Position und Ausrichtung des Anwenders fest und geben diese weiter an den Computer. Dieser wiederum aktualisiert anhand der neuen Daten die virtuelle Umgebung.

Um im dreidimensionalen Raum nützlich zu sein, müssen die Eingabegeräte die 6 Freiheitsgrade (6 DOF, degrees of freedom) berücksichtigen. Von diesen Freiheitsgraden sind drei durch die dreidimensionale Ausrichtung eines Körpers (X-, Y- und Z-Achse) gekennzeichnet. Die restlichen drei ergeben sich aus dem Abkippen eines Körpers nach links und rechts (roll), der Neigung eines Körpers nach vorn und hinten (pitch) und der Drehung eines Körpers nach links und rechts (yaw). [IM04]

2.2.1.1 SpaceMouse

Im Bereich der zweidimensionalen Anwendungen hat sich die handelsübliche Maus seit Jahren als Eingabegerät durchgesetzt. Eine Weiterentwicklung eben jener Maus sind die sogenannten „3-D-Mäuse“, wozu auch die SpaceMouse[2] gehört.

Die Sensorkappe lässt sich bewegen und drehen, womit sich Translationen und Rotationen auf allen 3 Raumachsen einfach realisieren lassen. Außerdem verfügt sie noch über 9 frei programmierbare Tasten. Siehe hierzu auch Abbildung 2.1 und Abbildung 2.2.

[2] Die SpaceMouse ist ein vom DLR (Deutsches Zentrum für Luft- und Raumfahrt) entwickeltes Eingabegerät für dreidimensionale Anwendungen.

Abbildung 2.1: DLR SpaceMouse aus [DLR04]

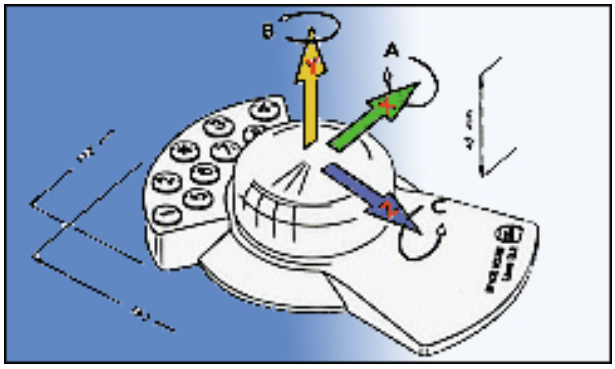

Abbildung 2.2: Funktionsweise der SpaceMouse aus [SPI04]

2.2.1.2 RingMouse

Als äußerst platzsparend erweist sich die sogenannte RingMouse, welche einfach mit einem Klettband um den Zeigefinger geschnallt wird (Abbildung 2.3). Das Gerät ist kabellos und schränkt die Freiheit der Hand kaum ein. Die Knöpfe dieser Maus lassen sich problemlos mit dem Daumen bedienen. Ein integrierter Ultraschallsender und üblicherweise am Monitor befestigte Ultraschallempfänger ermöglichen eine dreidimensionale Positionserfassung Abbildung 2.4). Die RingMouse läßt sich sowohl für zweidimensionale als auch für dreidimensionale Anwendungen verwenden. Allerdings empfiehlt es sich für Letztere eine Kombination mit einer 3D-Brille zu wählen, da das Auge lediglich zweidimensionale Informationen von einem Monitor wahrnimmt.

Abbildung 2.3: RingMouse aus [IM04]

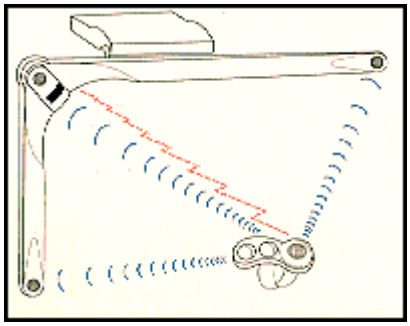

Abbildung 2.4: Positionserfassung der RingMouse aus [IM04]

2.2.1.3 Datenhandschuhe

Eine fortgeschrittenere Form der Eingabegeräte stellen die sogenannten Datenhandschuhe dar. 1987 entwickelten Jaron Lanier und Thomas Zimmermann den DataGlove, welcher als Urtyp aller Datenhandschuhe angesehen wird. Eine hochelastische Kunstfaser (Lycra[3]) bildet den Grundstoff für diesen Handschuh damit gewährleistet wird, das er sich bestmöglich an die individuellen Gegebenheiten einer Hand anpasst. An seinem Äußeren befinden sich fiberoptische Fasern, deren Lichtdurchlässigkeit abhängig von den Bewegungen der Finger ist. Die letztendliche Signalstärke einer jeden Faser wird gemessen und liefert dann die nötigen Daten um die exakte Position der Fingergelenke, bzw. der ganzen Hand zu berechnen. Zusätzlich verfügt der DataGlove noch über einen magnetischen Sensor, der die Position und Ausrichtung der Hand erfasst. Somit erhält der Computer alle nötigen Informationen für eine komplette Rekonstruktion der realen Hand.

Abbildung 2.5: Der DataGlove von VPL Research aus [IM04]

Heutzutage gibt es eine Vielzahl unterschiedlicher Datenhandschuhe. Dabei variiert die Funktionalität von der Genauigkeit der Gestenerkennung, über das Ertasten oder Fühlen eines Gegenstandes (Taktiles Feedback) bis hin zur Unterstützung von Kraftrückkopplung (Force Feedback). Dies alles bestimmt natürlich die Preise der verschiedenen Produkte.

[3] Wegen seiner hohen Elastizität wird Lycra in der Textilindustrie oft verwendet um ein Ausleiern der Textilien zu unterbinden.

2.2.1.4 DataSuit

Eine Weiterentwicklung des DataGlove, stellt der DataSuit dar. Hierbei handelt es sich um einen Ganzkörper-Datenanzug. Zimmermann und Lanier nutzen hierbei dieselben fiberoptischen Fasern wie bei ihrem Datenhandschuh. Bei diesem Anzug stellen der fehlende Tragekomfort und die immer wieder für jeden neuen Träger erforderlichen Kalibrierungseinstellungen bedeutende Nachteile dar. Die folgende Abbildung zeigt einen solchen Anzug im Einsatz.

Abbildung 2.6: Der DataSuit von VPL Research aus [LS03]

Inzwischen wird diese Technologie hauptsächlich als Motion Capture bei der Produktion von Computerspielen verwendet. Allerdings werden hierbei die Sensoren der Einfachheit halber direkt am Körper angebracht (vgl. Abbildung 2.7), anstatt einen dieser Anzüge zu verwenden.

Abbildung 2.7: Motion Capture aus [IM04]

2.2.2 Ausgabegeräte

Die Vorrichtungen zur Ausgabe haben die Aufgabe die menschlichen Wahrnehmungssinne zu stimulieren. Je besser dies geschieht, um so realer empfindet der Mensch die virtuelle Umgebung. [Fau97]

Bei der visuellen Ausgabe ist es deshalb immens wichtig, dem Benutzer einen räumlichen Eindruck, über die Umgebung in der er sich gerade befindet, zu verschaffen. Zunächst einmal muss deshalb der unterschiedliche Betrachtungspunkt der menschlichen Augen berücksichtigt werden (Augenparallaxe), schließlich entstehen dadurch kleine perspektivische Unterschiede in den beiden wahrgenommenen Bildern. Das VR-System muss also demnach ständig zwei unterschiedliche Bilder von ein und derselben Umgebung liefern. Und zwar einmal für das rechte und einmal für das linke Auge. Entsprechende Geräte weisen die verschiedenen Bilder dann dem zugehörigen Auge zu, wodurch ein wirklich dreidimensionaler Eindruck erzeugt wird. Diese Form der Darstellung wird als Stereodarstellung bezeichnet.

2.2.2.1 Polarisationsbrillen

Eine Möglichkeit zur Wahrnehmung von diesen sogenannten Stereobildern ergibt sich durch die Verwendung eines Videoprojektionssystems und Polarisationsbrillen. Wie der Name schon andeutet, finden hierbei Polarisationsfilter Verwendung. Ein solcher Filter ist ein optisch transparentes Medium, das, je nach Ausrichtung, nur Lichtwellen durchlässt, die auf einer bestimmten Ebene schwingen.

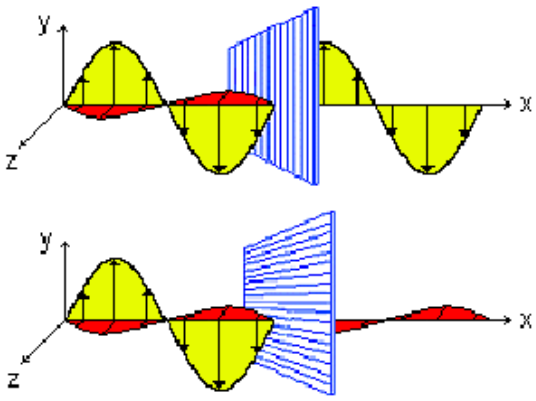

Abbildung 2.8: Lichtpolarisation mit Hilfe von Polarisationsfiltern aus [Gal03]

Bei diesem Projektionssystem kommen nun mindestens zwei Projektoren zum Einsatz, von denen jeder mit einem Polarisationsfilter ausgestattet ist. Der Computer berechnet und erstellt zunächst für jedes Auge ein unterschiedliches Bild der Szene, welches dann von einem der beiden Projektoren ausgesendet wird. Die ausgestrahlten Bilder werden nun durch die Filter orthogonal zueinander polarisiert. Dadurch entstehen letztendlich zwei Bilder mit unterschiedlicher Polarisation, wovon eines für das linke und das andere für das rechte Auge gedacht ist. Damit die Polarisation nicht verloren geht, ist eine spezielle Leinwand nötig. Mit einer normalen Wand ließe sich somit kein dreidimensionaler Effekt erzielen.

Der Benutzer trägt in diesem Fall besagte Polarisationsbrille, bei der wiederum, vor dem linken und dem rechten Auge, zwei orthogonal zueinander ausgerichtete Polfilter angebracht sind. Jene lassen nur einfallendes Licht einer bestimmten Polarisation hindurch und somit ist gewährleistet, das die perspektivisch unterschiedlichen Bilder auch das richtige Auge erreichen. Das menschliche Gehirn wird dadurch getäuscht und konstruiert dann einen räumlichen 3D-Effekt.

2.2.2.2 LCD ShutterGlasses

Diese LCD[4]-Brillen werden üblicherweise in Verbindung mit einem Monitor oder einem Projektor verwendet. Diese geben die unterschiedlichen Bilder für das linke und das rechte Auge im periodischen Wechsel aus. Anstelle der Brillengläser befinden sich bei den Shutter-Brillen zwei Glasscheiben zwischen denen sich Flüssigkristalle befinden. Die Ausrichtung dieser Kristalle kann nun mit Hilfe von Strom so verändert werden, das einfallendes Licht durchgelassen, oder aber auch blockiert wird. Die Shutter-Brille verfügt über einen eingebauten Infrarotempfänger, der es ermöglicht, sie mit der Ausgabe durch den Monitor bzw. Projektor zu synchronisieren. Der Computer sendet also, je nach aktuell dargestelltem Bild, ein Infrarotsignal an die Brille, welche daraufhin die Sicht für eines der beiden Augen verdunkelt und dem anderen Auge freie Sicht auf das ihm zugedachte Bild gewährt. Durch die Trägheit[5] des Auges entsteht dann die Illusion einer dreidimensionalen Ansicht.

---

[4] Liquid Crystal Display

[5] Aus gezeigten 30 Bildern pro Sekunde synchronisieren Auge und Gehirn eine flüssige Bewegung. [IM04]

Die folgende Abbildung 2.9 zeigt eine dieser LCD-Brillen.

Abbildung 2.9: LCD ShutterGlasses aus [IM04]

2.2.2.3 Head-Mounted-Display (HMD)

Ein HMD war das erste Gerät das dem Anwender das Gefühl vermittelte in eine andere Welt einzutauchen. Um dies zu erreichen ist die audiovisuelle Projektion der vom Computer erstellten virtuellen Welt auf Augen und Ohren notwendig, gleichzeitig aber auch die audiovisuelle Abschottung von der real existierenden Welt.

Bereits 1965 entwickelte Ivan Sutherland am Massachussets Institute of Technology (MIT) das erste Gerät dieser Art. Dieser Helm erhielt den Spitznamen „Damoklesschwert“, da er über ein unglaubliches Gewicht verfügte und deshalb niemals alleine und ohne weitere Hilfe zu tragen gewesen wäre. Aus diesem Grund wurde er von einem an der Decke befestigten Metallarm gehalten. [LS03]

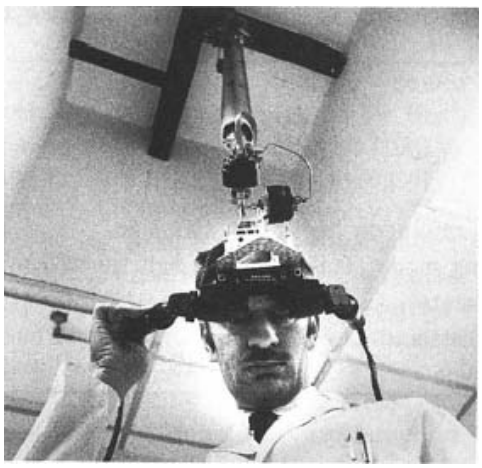

Abbildung 2.10: HMD von Sutherland aus [LS03]

Sutherland's HMD war der Grundstein für viele Weiterentwicklungen auf diesem Gebiet und somit gibt es zum jetzigen Zeitpunkt eine Vielzahl von HMD's. Grundsätzlich bestehen alle aus einem Helm, an dessen Vorderseite zwei Mini-LCD's und an dessen Seiten je eine Box angebracht wurde. Es gibt auch Versionen, bei denen anstatt der beiden LCD's zwei kleine Kathodenstrahlröhren (CRT) verwendet werden. Diese bieten kontrastreichere Bilder und eine wesentlich höhere Auflösung, allerdings sind sie auch nicht so leicht und vor allem viel teurer als die günstigen LCD's. Zusätzlich ist bei HMD's noch ein Ortungssystem, der sogenannte Polhelmus-Sensor, integriert, welches Position und Blickrichtung des Benutzers erfasst. Diese Informationen werden vom Computer in Echtzeit benötigt, um den korrekten Bildausschnitt zu berechnen und die beiden perspektivisch unterschiedlichen Bilder auf den Mini-LCD's des HMD's auszugeben.

Die verschiedenen Geräte lassen sich in zwei Gruppen unterteilen. Zum einen die Nonseethrough-HMD's (vgl. Abbildung 2.11) die eine volle Immersion in die virtuelle Welt gewährleisten und auf der anderen Seite die Seethrough-HMD's (vgl. Abbildung 2.12), welche der Augmented Reality zuzuordnen sind.

Bei den Nonseethrough - Modellen befinden sich die Displays senkrecht bzw. schräg vor den Augen und es besteht keine Möglichkeit durch sie hindurch zu schauen. Bei Seethrough – HMD's hingegen werden die Displays waagrecht über dem jeweiligen Auge montiert. Die Monitorbilder werden dann durch Vergrößerungslinsen auf einen halbversilberten Spiegel, welcher wiederum im 45° Winkel vor dem Auge angebracht wird, projiziert. Mit diesem System kann der Benutzer nicht nur die reflektierten Bilder, sondern auch die ihn umgebende Außenwelt sehen.

Abbildung 2.11: Nonseethrough - HMD aus [Bei04]

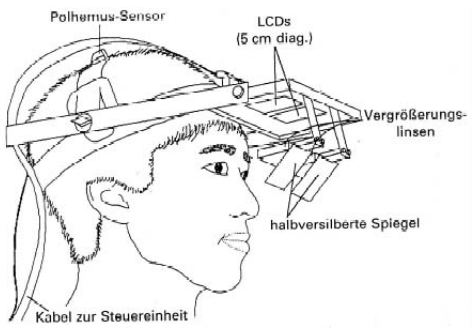

Abbildung 2.12: Funktionsprinzip eines Seethrough - HMD aus [LS03]

2.2.3 Kombinierte Geräte

Hierbei handelt es sich entweder um Geräte die sowohl zur Eingabe als auch zur Ausgabe dienen, oder um Lösungen bei denen bestimme Kombinationen aus Eingabe- und Ausgabegeräten verwendet werden.

2.2.3.1 Binocular Omni-Orientation Monitor (BOOM)

Dies ist ein stereoskopisches Sichtgerät, das am Kopf geführt wird. Ein sogenanntes Head-Coupled-Display (HCD). Die virtuelle Umgebung wird in diesem Fall innerhalb einer kleinen Box angezeigt. Durch zwei kleine Öffnungen kann der Anwender die generierte Szene betrachten. Die Anzeigebox wiederum ist an einem beweglichen Schwenkarm angebracht, wodurch, mit Hilfe von Sensoren, Bewegungen innerhalb der virtuellen Szene ermöglicht werden. Der Benutzer schaut also in die Box und kann sich dann innerhalb des Aktionsraums des Geräts frei in der künstlichen Welt bewegen.

Abbildung 2.13: BOOM aus [FSL04]

2.2.3.2 Cave Automatic Virtual Environment (CAVE)

CAVEs wurden von der University of Illinois entwickelt. Dabei handelt es sich meistens um einen quadratischen Raum, an dessen Wände und auf dessen Boden Stereobilder projiziert werden. Die Kopfposition des Benutzers wird erfasst und die generierten Bilder seiner Blickrichtung entsprechend angepasst. Dadurch entsteht das Gefühl in eine andere Welt einzutauchen. Dieser Eindruck kann durch einen beweglichen Boden noch verstärkt werden, da sich so auch noch realistische Vibrationen des Untergrunds erzeugen lassen.

Innerhalb der CAVE werden in der Regel synchronisierte Shutter-Brillen verwendet um die dreidimensionale Welt wahrzunehmen. Zur Eingabe eignet sich ein Datenhandschuh, allerdings finden auch andere Eingabegeräte dort oft Verwendung.

Jedoch hat der Anwender bei diesem System immer noch Bezugspunkte zur realen Welt. Zum einen sind die Ecken des Raumes als störende Ungleichmäßigkeit noch immer wahrnehmbar und zum anderen ist die Bewegungsfreiheit durch die Raumgröße festgelegt.

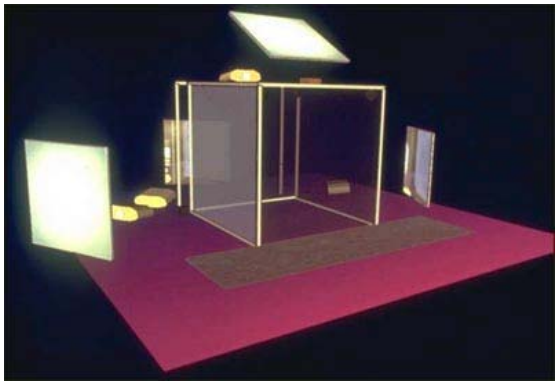

Abbildung 2.14: CAVE aus [IM04]

2.2.3.3 Immersive Continuous Environement (i-CONE)

Die i-CONE wurde vom Fraunhofer Institut für Medienkommunikation (IMK) in Zusammenarbeit mit der BARCO GmbH aus Belgien entwickelt. Es handelt sich hierbei um ein Projektionssystem, bei dem die virtuelle Umgebung auf eine große horizontale Leinwand, die in einem Winkel von bis zu 230° gewölbt ist, abgebildet wird. Vier Projektoren finden hier Verwendung und deren Teilbilder werden mit Hilfe der integrierten Edge-Blending-Technologie zu einem nahtlosen, stereoskopischen Bild vereint. Im Gegensatz zur CAVE , wo die Technik der Rückprojektion eingesetzt wird, kommt bei der i-CONE eine Frontprojektionstechnik zum Zuge. Daraus resultiert, neben einer effektiveren Raumausnutzung, eine deutlich bessere Bildqualität (Kontrast, Sättigung). Durch die geringeren Platzanforderungen ist es größeren Gruppen zum ersten Mal möglich, Virtual Reality-Anwendungen gemeinsam zu erleben. [IMK04]

Der große Vorteil dieses Systems ist das Fehlen von Ecken und Kanten, dadurch werden CAVE-typische Verzerrungs- und Reflektionseffekte vermieden. Eine weitere Besonderheit ist die hohe Auflösung von 5760 x 1320 Pixel, wodurch qualitativ hochwertige Bilder entstehen. Desweiteren verfügt die i-CONE über ein hochwertiges 3-D Audiosystem mit bis zu 24 Kanälen. [IMK04]

Um das stereoskopische Bild korrekt wahrzunehmen, trägt der Benutzer eine Shutter-Brille. Für die auditive Wahrnehmung sind dank des Audiosystems keine weiteren Hilfsmittel nötig.

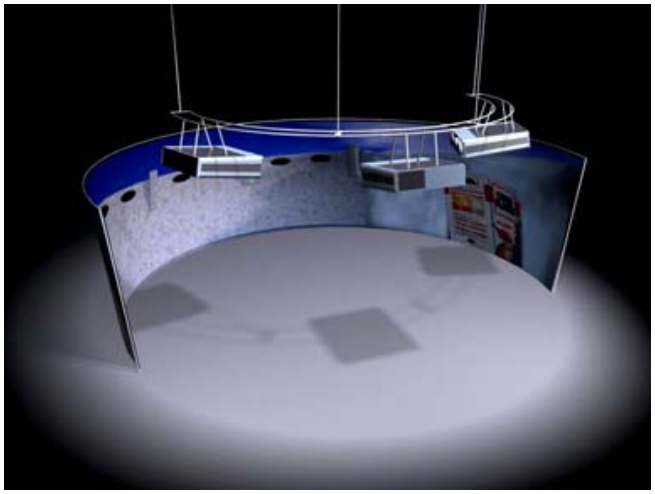

Abbildung 2.15: i-CONE aus [IMK04]

2.3 Verschiedene Arten der Virtual Reality

Wie zuvor schon erwähnt wurde, ist Virtual Reality nicht gleich Virtual Reality. Je nach verwendeter Hard- und Software kann die VR unterschiedliche Formen, welche im Folgenden kurz angesprochen werden, annehmen.

2.3.1 Desktop VR

Dies ist vermutlich die einfachste und häufigst verwendete Form, da keine spezielle Hardware benötigt wird. Die typische Konfiguration besteht also lediglich aus einem PC mit Soundkarte und Monitor. Tastatur, Maus oder Joystick dienen zur Eingabe. Diese Darstellungsform wird auch als „Window on World Systeme“ (WoW) bezeichnet.

2.3.2 Fish-Tank VR

Hierbei handelt es sich um Desktop VR bei der ein Tracking-System verwendet wird, um die Position des Anwenders zu bestimmen. Außerdem finden hier Shutter-Brillen ihren Einsatz, um stereoskopische Bilder am Monitor wahrnehmen zu können. Die dargestellten Szenen werden der Position des Benutzers entsprechend angepasst.

2.3.3 Videogebundene VR / Video Mapping

Auch ein WoW-System. In diesem Fall wird der Anwender mit Hilfe von Videokameras aufgezeichnet und kann sich selbst dann, auf einem Bildschirm, innerhalb der virtuellen Umgebung betrachten und mit dieser interagieren.

2.3.4 Telepräsenz

Telepräsenz umfaßt die Möglichkeit, mit Hilfe von geeigneten Technologien, an verschiedenen Orten virtuell anwesend zu sein. Die Sinne eines menschlichen Operators werden mit einem Sensor irgendwo in der realen Welt verknüpft. Dadurch erhält der Anwender Einblick auf die Umgebung, in der sich der Sensor befindet und kann dann anhand dieser Daten agieren. Dies kann mit den unterschiedlichsten Ein- und Ausgabegeräten realisiert werden.

Beispielsweise beim Entschärfen von Sprengsätzen könnte, aufgrund des hohen Risikos für Menschen, ein ferngesteuerter Roboter eingesetzt werden, der mit Videosensoren ausgestattet ist. Oder auch bei medizinischen Operationen könnten kleine, mit einer Videokamera ausgerüstete, Instrumente verwendet werden.

2.3.5 Immersive VR

Bei der immersiven VR wird versucht, dem Benutzer das Gefühl zu vermitteln, vollkommen in eine andere Welt einzutauchen. Wie zuvor (in Abschnitt 2.2.2.3) schon kurz erwähnt, wird versucht dies zu erreichen, indem man Gesichts- und Gehörsinn des Anwenders bestmöglich von der realen Welt abschottet. Gleichzeitig muss das System aber auch andauernd genau diese beiden Sinne stimulieren.

Diese VR - Systeme werden oftmals mit einem HMD realisiert. Als Alternative dazu bietet sich eine CAVE an.

2.3.6 Mixed Reality / Augmented Reality

Mixed Reality schließlich stellt eine Kombination aus wirklicher und virtueller Welt dar. Dies ist besonders dann sinnvoll, wenn der Benutzer die reale Umgebung stets wahrnehmen und die künstliche Umgebung nur zusätzliche Informationen liefern soll. Dadurch können viele Aufgaben die sich dem Anwender stellen, vereinfacht werden.

Im Laufe der neunziger Jahre entstanden viele dieser vermischten Umgebungen unter der Bezeichnung „teil-immersiv“. Je nach Größe der realen oder virtuellen Anteile reicht das Spektrum hierbei von einer realen Umgebung mit geringfügigen virtuellen Elementen bis hin zu einer fast ausschließlich virtuellen Umgebung mit wenigen realen Elementen. Die ganzen Kombinationsmöglichkeiten werden unter dem Begriff Mixed Reality zusammengefasst.

Um eine MR/AR-Umgebung darzustellen, wird üblicherweise auf, die weiter oben in Abschnitt 2.2.2.3 auch schon beschriebenen, Seethrough-HMD’s zurückgegriffen.

2.4 Anwendungsgebiete

Wo lässt sich Virtual Reality sinnvoll einsetzen? Und warum? Die größten möglichen Vorteile durch den Einsatz von VR liegen in der hohen Transparenz und Interaktivität. Dadurch können Probleme viel schneller erfasst und verstanden werden. Und somit lassen sich diese Probleme wahrscheinlich auch viel schneller lösen. Demzufolge verfügt die VR über ein gewaltiges Potential, welches auch schon in viele Bereiche Einzug gehalten hat und deren Entwicklung vorantreibt. Sei es in der Medizin, Unterhaltung oder bei der Architektur zur Rekonstruktion von historischen Bauten. Im folgenden nun eine Beschreibung einiger Anwendungsgebiete.

2.4.1 Architektur / Bauwesen

Um Entwurfs- und Konstruktionsfehler möglichst frühzeitig zu entdecken, werden hier gesamte Bauprojekte mit Hilfe von CAD[6]-Werkzeugen modelliert und visualisiert, noch bevor die eigentlichen Bauarbeiten begonnen haben.

Mit Hilfe eines VR-Systems und einer Datenbank kann ein solches Bauvorhaben deutlich optimiert und für den Betrachter erheblich transparenter dargestellt werden. Vor allem im Bezug auf die gesetzlichen, finanziellen, technischen, ökonomischen und ökologischen Rahmenbedingungen des Projekts. Dies ermöglicht eine hohe Interak-

[6] CAD steht für Computer Aided Design

tivität wodurch Architekt, Bauherr und Mieter sozusagen schon in der Entwurfsphase mit einbezogen werden und gemeinsam am Projekt arbeiten.

Die folgende Abbildung zeigt zum Beispiel ein VRML-Objekt des Instituts für Informatik der Universität Zürich.

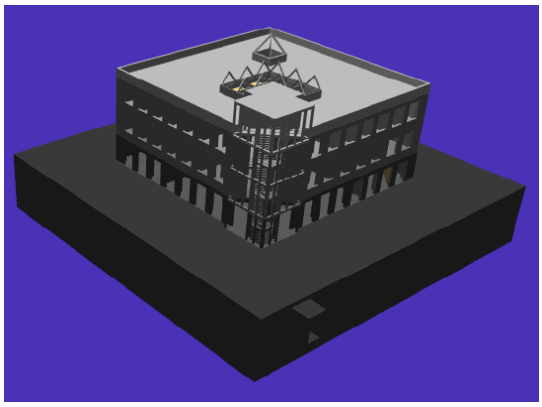

Abbildung 2.16: computergeneriertes Gebäudemodell aus [KRS98]

Dieses Modell wurde zunächst mit einem 3D-Modellierwerkzeug konstruiert und dann ins VRML-Format konvertiert. Noch bevor das eigentliche Institutsgebäude fertiggestellt worden war, konnte dieses 3D-Modell im virtuellen Raum für Begehungen verwendet werden [KRS98].

Desweiteren kann auch die Innenarchitektur auf diese Art simuliert werden. So kann z. B. der Mieter, schon bevor das eigentliche Haus fertig ist, verschiedene Produkte von Möbelhäusern in seinem virtuellen Haus platzieren, bzw. damit experimentieren bis er sich die gewünschte Einrichtung zusammengestellt hat.

Als schwierig stellte es sich in der Vergangenheit dar, die Raumakustik eines Gebäudes korrekt zu berechnen und darzustellen. Z. B. für eine Konzerthalle. Allerdings ist es dank neuer Technologien inzwischen möglich, digitale Signale so zu verarbeiten, dass eine Echtzeitsimulation akustischer Faktoren möglich wird. [LS03]

Solch ein Modell kann nicht nur von Häusern erstellt werden, sondern auch für z. B. Flugzeuge, Schiffe usw. Und durch die angewendeten Verfahren kann eine erhöhte Qualitätssicherung und unter Umständen eine enorme Kosteneinsparung bei diesen Projekten erzielt werden.

2.4.2 Medizin

Im medizinischen Bereich wird VR immer häufiger integriert und ständig weiterentwickelt. Besonders im Bezug auf die Visualisierung der menschlichen Anatomie. Auch hier können die medizinbezogenen Informationen anschaulich dargestellt werden und sind somit sowohl für die behandelnden Ärzte als auch für die Patienten besser zugänglich. In Kombination mit einem Client/Server-System können z. B. wichtige Informationen für die Diagnose oder Operationsplanung über das Internet ausgetauscht oder angefordert werden. Was wiederum die Behandlung wesentlich vereinfacht und verkürzt.

Betrachten wir z. B. die Chirurgie. Hier liefert das Endoskop mithilfe von winzigen Kameras Bilder aus dem Inneren des Körpers. Diese Daten werden dann zweidimensional auf einem Bildschirm dargestellt. Für den Operierenden wäre es besser, oder leichter, wenn ihm dreidimensionale Bilder zur Orientierung zur Verfügung ständen. Dies ist mit einem 3D-Video-Endoskop, welches z. B. von der Opticon Gesellschaft für Optik und Elektronik in Karlsruhe oder dem Frauenhofer-Institut für Produktionstechnik und Automatisierung (IPA) in Stuttgart angeboten wird, möglich. [LS03]

Es gibt auch zahlreiche Simulationsmöglichkeiten von medizinischen Eingriffen oder der menschlichen Anatomie um angehende Mediziner auszubilden, oder aber auch für Weiterbildungen.

2.4.3 Unterhaltung

Seit Ende der Achtziger hat sich der Markt für Computerspiele ständig weiterentwickelt und vergrößert. Alljährlich werden leistungsstärkere Systeme für Videospiele verfügbar, da die Unterhaltungsbranche Unsummen investiert um im Konkurrenzkampf mit anderen Anbietern den Kunden immer bessere Spiele zu bieten. Hierbei finden auch vielerlei VR-Technologien ihre Anwendung.

Bezogen sich diese Computerspiele zunächst nur auf einzelne Benutzer, so ergibt sich dank des Internets inzwischen die Möglichkeit, etliche Spieler innerhalb einer Anwendung zu vereinen. Dies bietet die Grundlage um z. B. gemeinsam Abenteuer zu beste-

hen oder um sich bei einem Wettbewerb in Teams zu beweisen. Aus diesen Möglichkeiten entstanden auch schon ganze Spielergemeinschaften, die auch noch außerhalb der eigentlichen Spiele in Kontakt bleiben. Allein dies, verdeutlicht die große Nachfrage an dieser Form der Freizeitgestaltung.

Die folgende Abbildung zeigt VRML-Dungeon, eine Spielumgebung bei der der Benutzer eine 3D-Figur durch ein ebenfalls dreidimensionales Labyrinth steuert und dabei Rätsel lösen muss.

Abbildung 2.17: VRML-Dungeon aus [KRS98]

2.4.4 Unterricht / Schule

Da Schüler Lehrstoffe umso besser aufnehmen, wenn sie in den Unterricht integriert werden, verwenden immer mehr Lehrer Computer zur Gestaltung ihrer Stunde. Dadurch lassen sich virtuelle Umgebungen realisieren in denen sich mathematische oder physikalische Sachverhalte viel besser als in jedem Schulbuch visualisieren lassen.

Desweiteren besteht die Möglichkeit zur Interaktion mit diesen Umgebungen und die Kinder können durch ein wenig experimentieren von alleine auf die Lösungen zu ihren Fragen kommen. Durch diese eigenen Erfolge werden die Schüler viel höher motiviert und die Lernerfolge steigen.

Kapitel 3

Persistenz

“Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created).”

Grady Booch (1991)

Dieses Kapitel soll einen Überblick über den Begriff „Persistenz“ vermitteln. Zunächst erfolgt eine Begriffserklärung und abschließend eine Beschreibung verschiedener Ansätze zur Realisierung von Persistenz.

3.1 Was ist Persistenz?

Der Begriff Persistenz wird hauptsächlich in der objektorientierten Programmierung dazu verwendet um die Fähigkeit eines Objekts, bzw. einer Datenstruktur, zur Speicherung zu kennzeichnen. Durch diese Eigenschaft erhöht sich die „Lebensdauer“ der Objekte die in der eigentlichen Anwendung verwendet werden. Möglich gemacht wird dies dadurch, das die Daten aus „flüchtigem“ Speicher[7] auf beständige Speichermedien geschrieben werden. Andernfalls, ohne jegliche Persistenz-Funktionen, gingen diese Datenstrukturen verloren, sobald das Programm beendet wird. Solch vergängliche Daten werden auch als transient bezeichnet.

Und natürlich muss es auch möglich sein, anhand der, auf nun beständigen Speichermedien, hinterlegten Daten, die zuvor gesicherten Objekte innerhalb der Anwendung wieder herzustellen.

[7] „Flüchtiger“ Speicher wäre zum Beispiel der RAM, welcher zum Einen nicht für dauerhafte Speicherung ausgelegt ist und zum Anderen oftmals auch nicht über die nötigen Kapazitäten verfügt.

Ursprünglich wurde der Begriff in den späten 70’er Jahren von Malcolm P. Atkinson geprägt. Schon damals forderte er einen, bzw. suchte selber nach einem Weg, bei dem die komplette Datenstruktur persistent bleibt zwischen zwei Programmausführungen. Diese Notwendigkeit beruht auf der nun folgenden Erklärung.

Jegliche Daten kann man in zwei Kategorien einordnen. Auf der einen Seite in „kurzlebige“ Informationen, welche nur während der Programmausführung existieren und auch nur von der Anwendung selbst bearbeitet werden und auf der anderen Seite in „langlebige“ Daten, die wiederum nur vom Dateisystem oder von der Datenbank manipuliert werden.

Eine feinere Unterteilung liefert das Persistenz-Spektrum aus [ABC83]. Dort werden folgende Kategorien definiert:

1. „flüchtige“ Ergebnisse die durch die Auflösung eines Ausdrucks entstehen
2. lokale Variablen innerhalb einer Funktion
3. eigene oder globale variablen und heap items, die ihren Geltungsbereich überschreiten
4. Daten die auch zwischen den Programmausführungen existieren
5. Daten die zwischen verschiedenen Programmversionen existieren
6. Daten die das Programm „überleben“

Logischerweise beziehen sich die ersten drei Punkte auf die Anwendung selbst und ihre Ausführung, gehören also zu der Gruppe der „kurzlebigen“ Daten. Während die letzten drei Punkte ein gewisses Speicherungssystem voraussetzen und dadurch „langlebig“ werden. Sei das nun in Form einer Datenbank oder einfachen Dateien. Allerdings haben sich bei den Punkten 4 und 5, hauptsächlich Systeme durchgesetzt, die ihre Daten in einfache Files speichern.

Nun stellen aber sowohl Anwendung als auch Datenbank dieselben Daten auf unterschiedliche Weise dar. Ein Programm verwendet z. B. Arrays, Records, Sets und abstrakte Datentypen, wohingegen eine Datenbank auf einem relationalen oder hierarchischen Datenmodell basiert. Deshalb ist es stets nötig, die Informationen zwischen den beiden Darstellungsformen umzuwandeln. Dabei wird unter Umständen und je nach

Datenmenge viel Rechenkapazität, Speicherplatz und Zeit verbraucht. Außerdem geht auch Arbeitszeit verloren, da die Entwickler sich bei jedem Programm um diese Umwandlung kümmern und diese auch bei Programmänderungen stets aktuell halten müssen. Im schlimmsten Fall könnte durch den zusätzlichen Code die Funktionalität oder die Leistung der eigentlichen Anwendung negativ beeinträchtigt werden. [ABC83]

Ein weiterer großer Nachteil dieser unterschiedlichen Darstellungsformen ist der Verlust des Schutzes für die einzelnen Datentypen während der Speicherung. Bei der Umwandlung durch das Programm wird die Datenstruktur nicht mehr geschützt und kann mitunter verloren gehen.

Deshalb war Atkinson damals der Ansicht, dass die Persistenz eine Eigenschaft der eigentlichen Datenstruktur sein sollte. Denn dann würde ein erheblicher Programmieraufwand und eine große potentielle Fehlerquelle entfallen.

3.2 Orthogonale Persistenz

Unter orthogonaler Persistenz versteht man eigentlich nur ein persistentes System das folgende Kriterien erfüllt:

- *Persistenzunabhängigkeit*
  Die Persistenz eines Datenobjekts ist unabhängig davon, wie das Programm dieses Objekt verändert. Und umgekehrt werden Programmteile auch unabhängig von der Persistenz der Daten, die sie verarbeiten, dargestellt. Zum Beispiel soll eine Funktion mit persistenten oder aber auch transienten Funktionsparametern aufrufbar sein.
- *Typ Orthogonalität*
  Alle Datenobjekte haben die gleichen Persistenzmöglichkeiten, völlig unabhängig von ihrem Typ.

- *Transitive Persistenz*
  Alle Objekte, die von einem persistenten Objekt erreicht werden könnten, müssen mitgespeichert werden. Da die potentielle Möglichkeit besteht, das bei einer späteren Programmausführung auf eines dieser Objekte zugegriffen wird. Hierbei können auch transiente Objekte persistent werden.

3.3 Persistenztechniken

Es gibt verschiedene Ansätze um persistente Datenstrukturen zu realisieren. Diese unterschiedlichen Verfahren führen allerdings nicht unbedingt zu gleichwertigen Ergebnissen, bzw. sie erfüllen nicht alle die notwendigen Kriterien für eine orthogonale Persistenz.

In [CO95] legen sich die Autoren auf vier unterschiedliche Grundtechniken, um die Lebenszeit von Datenstrukturen zu erhöhen ohne die Informationen auf traditionelle Weise in einer anderen Form zu speichern, fest. Die dort beschriebenen Techniken lauten wie folgt:

3.3.1 checkpointing

Hierbei wird der aktuelle Zustand einer laufenden Anwendung komplett gespeichert. D. h. wirklich alles was von der Anwendung verwendet, manipuliert oder erzeugt wurde, wird persistent gemacht. Auch Prozesse bzw. Threads. Daraus ergibt sich, das es keinerlei Möglichkeiten für Entwickler gibt, in irgendeiner Form festzulegen, welche Informationen persistent werden sollen und bei welchen Informationen dies unnötig ist.

Desweiteren besteht ein großer Nachteil darin, das sich dieses Verfahren nicht für konkurrierende Zugriffe auf die Daten durch mehrere Programminstanzen eignet, da in solch einem Fall zwangsläufig unterschiedliche Zustände als ein Zustand gespeichert würden. Weiterhin führt bei dieser Vorgehensweise ein Update des Programms zwangsläufig zu Inkompatiblitäten mit zuvor gespeicherten Daten. Dadurch werden jene Informationen nutzlos. Und zu guter Letzt werden auch etliche Daten gespeichert, bei denen

eigentlich daraus gar kein Nutzen für die weitere oder für die nächste Programmnutzung entsteht. Viele dieser Informationen sind sozusagen völlig überflüssig und erhöhen nur den Aufwand.

Da hierbei konsequenterweise alles gespeichert wird, werden logischerweise alle Datentypen gesichert und somit wird hier Typ Orthogonalität unterstützt. Allerdings trifft dies nicht auf die Persistenzunabhängigkeit zu, da ja alles andauernd persistent gemacht wird und es somit gar keine transienten Daten gibt.

3.3.2 persistence by property

„persistence by property“ bedeutet, das die Anwendung darüber entscheidet welche Informationen gespeichert und welche nicht gespeichert werden sollen. Diese Auswahl kann auf unterschiedliche Art erfolgen. Sei es durch bereits im Quellcode des Programms festgelegte persistente Datentypen, oder aber auch durch dynamische Vorgaben während der Laufzeit der Anwendung.

Dadurch das dieses Verfahren die Daten auf eine bestimmte Art und Weise sortiert und daraufhin oftmals unterschiedlich verwendet, wird bei diesem Ansatz die Persistenzunabhängigkeit vernachlässigt.

3.3.3 persistence by copying

Bei dieser Technik wird eine Datenstruktur durchschritten, in eine vereinfachte Form umgewandelt und dann auf nicht-flüchtigen Medien gespeichert. Dabei enthalten die gesicherten Informationen nicht nur die Daten selbst, sondern auch noch die zugrundeliegende Struktur. Somit kann später beim Einlesen der Informationen die zuvor vorhandene Datenstruktur rekonstruiert werden. Dies wird auch als „data flattening“ oder „pickling“ bezeichnet.

Das Hauptproblem dieses Mechanismus besteht allerdings darin, das innerhalb der gespeicherten Daten keine Informationen bezüglich der Objektidentitäten enthalten

sind. Dies führt dazu, das möglicherweise überflüssige Instanzen von Objekten erzeugt werden und diese auch nicht mehr im richtigen Verhältnis zueinander stehen.

Nehmen wir z. B. ein Programm welches zwei Objekte A und B erzeugt. Diese beiden enthalten nun wiederum je einen Zeiger auf das Objekt C. Wird nun A gespeichert, dann wird eine Datei erzeugt, die die Objekte A und C enthält. Speichert man B, entsteht folglich eine Datei mit den Objekten B und C. Es entstehen also zwei Dateien wodurch C auch zweimal gesichert wurde. Und da C innerhalb der Dateien über keine Identitätsinformationen verfügt, wirkt sich dies wie folgt aus: Sobald nun in einer späteren Programmausführung versucht wird die beiden Dateien wiederherzustellen, entstehen dabei insgesamt vier Objekte. Und zwar A, B und zwei voneinander unabhängige C - Objekte. Da diese unabhängig sind, verfügen A und B auch nicht mehr über korrekte Zeiger auf ein Objekt, das sie sich vorher geteilt haben.

3.3.4 persistence by reachabilty (Transitive Persistenz)

Wie schon bei der orthogonalen Persistenz beschrieben, werden hierbei alle Objekte gespeichert, die von einem persistenten Datenobjekt erreicht werden können. Eben weil eine spätere Programmausführung unter Umständen auf diese Objekte zugreifen könnte.

Betrachtet man nun diese unterschiedlichen Vorgehensweisen, so kommt im Rahmen dieser Diplomarbeit eigentlich nur eine Umsetzung der transitiven Persistenz in Frage. Denn nur durch solch ein System lässt sich garantieren, das alle Objekte, die für eine virtuelle Szene relevant sind, gespeichert werden. Und dies ohne zeitgleich überflüssige Daten zu sichern.

Kapitel 4

Virtual Reality Modeling Language (VRML)

“The aim of VRML is to empower programmers to build a world of interlocking applications which exploit the technological convergence, which has made a new kind of application possible: virtual socialenvironments.”

[Roc99]

In diesem Kapitel erfolgt nun eine kleine Einführung in VRML. Nach einer kurzen Einleitung erfolgt die Entstehungsgeschichte und anschließend die Funktionsweise von VRML. Als nächstes werden die Persistenzmöglichkeiten innerhalb der Virtual Reality Modeling Language angesprochen, bevor auf die neueste Erweiterung namens eXtensible 3D eingegangen wird. Zum Ende des Kapitels erfolgt dann noch eine Bewertung bezüglich des Nutzens von VRML im Zusammenhang mit dem praktischen Teil dieser Diplomarbeit.

Die Virtual Reality Modeling Language ist kein VR-System im eigentlichen Sinne, sondern vielmehr eine Sprache zur abstrakten Beschreibung von virtuellen Welten. Einerseits wird VRML zur Generierung dreidimensionaler Webseiten und andererseits als Austauschformat verwendet. Gerade Letzteres ist interessant. Es bietet einfache Beschreibungsmöglichkeiten der üblichen Strukturen, wie z. B. Transformationen, Geometrien und Lichtquellen. Dadurch eignet sich VRML zum Austausch der 3D-Daten zwischen unterschiedlichen Modellierungs-, bzw. Entwicklungsapplikationen.

4.1 VRML 1.0

Bei der ersten VRML-Version musste sich zunächst einmal darauf geeinigt werden, wie man Objekte, und vor allem deren Eigenschaften, in einem dreidimensionalen Raum so beschreibt, das sie möglichst kompatibel sind. Schließlich sollten sie auch in völlig unabhängig voneinander entwickelter Darstellungssoftware ähnlich dargestellt werden.

Der von Silicon Graphics entwickelte 3D-Standard „Open Inventor“ wurde 1995 modifiziert und als VRML 1.0 übernommen. Dadurch wurden folgende Punkte verbindlich festgelegt [KRS98]:

- die Nutzung des Koordinatensystems
- eine Syntax, mit der Objekte aus Polygonen zusammengesetzt und mit Attributen wie Farbe und Oberflächenbeschaffenheit (glänzend oder matt, transparent oder opak) versehen werden
- die Platzierung der Objekte in Szenen
- auch solche Objekte, die nicht lokal, sondern irgendwo im Netz gespeichert sind
- die Verwendung alternativer Darstellungen, z.B. für unterschiedliche Detailstufen (Levels of Detail) oder für verschiedene Situationen (Miniaturdarstellung oder unterschiedliche grafische Auflösungen)

Da HTTP als Kommunikationsprotokoll im Internet wunderbar funktionierte, entschied man sich dafür daran festzuhalten. Schließlich wäre eine Neuentwicklung überflüssig und würde nur zu einem erhöhten Arbeitsaufwand führen. Im Laufe der Zeit wurde VRML somit auch zunehmend als Erweiterung von HTML angesehen, zumal die führenden Hersteller von VRML-Browsern damit begannen ihre Produkte als plug-ins für HTML-Browser anzubieten.

Mit steigender Anzahl an VRML-Anwendungen kamen aber auch immer wieder neue Mängel und Schwächen von VRML 1.0 ans Tageslicht. Es fehlten eine Vielzahl von elementaren Möglichkeiten. Vor allem bei der Beschreibung der physischen Eigenschaften von Objekten. Auch Funktionen für die Verwirklichung von Animationen und Interaktion oder die Einbindung von Multimedia-Daten waren nicht vorhanden.

4.2 VRML 2.0 (VRML97)

Ein Jahr später formulierte man ein Ausschreiben für die 2. Version von VRML. Insgesamt wurden sechs formelle Vorschläge eingereicht und ausgiebig diskutiert. Am Ende setzte sich schließlich, das von Sony und Silicon Graphics entwickelte Konzept, „Moving Worlds“ in einer Internetabstimmung als neuer Standard durch. [Pes97]

Die wichtigsten Neuerungen waren [KRS98]:

- *Syntax-Bereinigung*
  Die zentrale VRML-Datenstruktur wurde neu geordnet und die meisten syntaktischen Problemstellen in VRML 1.0 behoben.
- *Neue Inhalte: Ton, Animation*
  Analog zu dem Beleuchtungsmodell in VRML 1.0 wurde ein Model für den Umgang mit verschiedenen Tonquellen geschaffen, sowie verbesserte Schnittstellen für die Behandlung von Multimedia und anderen zeitbedingten Interaktionen.
- *Erweiterbarkeit (1): Scripts*
  Neue Mechanismen, die das Ausprobieren neuer Schnittstellen möglichst einfach gestalten und zeitgleich die Notwendigkeit für Änderungen des Standards minimieren sollten, wurden integriert.
- *Erweiterbarkeit (2): PROTOs, Scripts*
- *Erweiterbarkeit (3): EAI*
  Die Definition eines External-Authoring Interface, das VRML eng mit Java integriert. Damit wurde die 1.0-Syntax enorm erweitert und alle möglichen Anwendungsinhalte würden als VRML-Modelle realisiert werden können. Allerdings war die Definition noch nicht vollständig.
- *Optimierung: BCF*
  Auch nicht ganz fertig, aber neue Mittel um VRML’s platz- und rechenzeitraubendes ASCII-Format um ein optimiertes Binärformat sowie eine weitere zehnfache Komprimierung der Geometrie zu erweitern.

Im September 1997 wurde VRML 2.0 nach einigen kleineren Änderungen als VRML97 durch die International Standards Organization (ISO) als Industriestandard ISO/IEC 14772 festgesetzt. [Eib01]

Trotz alledem war VRML noch längst nicht komplett und verfügte über einige empfindliche Mängel [KRS98]:

- *Die starke Begrenzung der „Welt“*
  Die gesamte Szene wird innerhalb einer einzigen Quellcode-Datei definiert, dadurch fehlt die Grundlage für wirklich skalierbare Szenendefinitionen.
- *Mangelhaftes Objektkonzept*
  Es gibt keine Möglichkeit zur objektorientierten Programmierung von Klassen mit vererbbaren Funktionen.
- *Die eigenartige interne Kommunikation*
  VRML kennt keine Typed Messages und bietet keinerlei Möglichkeiten, den Sender eines Events zu ermitteln und somit folglich ihn zu überprüfen.
- *Die kaum vorhandene Datensicherheit*
  VRML97 ist noch für eine Single-User-Umgebung konzipiert. Weder interne Sprachmittel noch externe Schnittstellen werden angeboten, mit denen sich ein Programm gegen die Auswirkungen anderer Programme effektiv schützen könnte. Die Autoren von VRML97 vertreten den Standpunkt, ein solcher Schutz sei die Aufgabe des Betriebssystems bzw. des Netzes.

4.3 Funktionsweise

VRML ist eine deklarative Sprache. Daraus folgt das man innerhalb eines VRML-Skriptes lediglich die Eigenschaften der virtuellen Welt definiert, anstatt sich, wie bei einer prozeduralen Sprache, mit allen Einzelheiten des Programmes zu beschäftigen. Die VRML-Dateien werden auch nicht erst kompiliert und dann ausgeführt, sondern zur Laufzeit analysiert. Dies liefert dann die nötigen Informationen um die Szene zu erstellen.

Man braucht sich also z. B. nicht darum zu kümmern, wann und wo eine Eingabe erfolgt ist oder wie ein neues Fenster geöffnet werden soll. Stattdessen legt man wie gesagt nur die Eigenschaften fest, darunter fallen unter anderem die Positionen und Ausrichtungen von Objekten, deren Erscheinungsbild oder aber auch was passieren soll, wenn sie angeklickt werden.

Innerhalb der Syntax von VRML spricht man von Knoten, die im Grunde Objekten entsprechen. Diese Knoten haben dementsprechend auch unterschiedliche Felder. Eine besondere Form der Knoten stellen Sensoren und Interpolatoren dar. Sensoren werten äussere Einflüsse (z. B. eine Eingabe durch den Anwender oder Zeitintervalle) aus und erzeugen Events, welche dann wiederum ausgesendet werden. Interpolatoren hingegen empfangen Events, werten diese aus und erzeugen wiederum neue Events.

Jegliche Knoten, also auch Sensoren und Interpolatoren, werden hier in einem Szenengraphen[8] ausgehend vom Nullpunkt angeordnet. Der Nullpunkt steht für die gesamte Szene, den Ursprung der Welt sozusagen. Außerdem enthält der Szenengraph auch noch sogenannte Routen. Diese dienen dazu, Felder die Events auslösen mit Feldern die Events empfangen zu verbinden. Somit werden also Ereignisse übertragen. Die folgende Abbildung 4.1 zeigt einen einfachen VRML-Szenengraph.

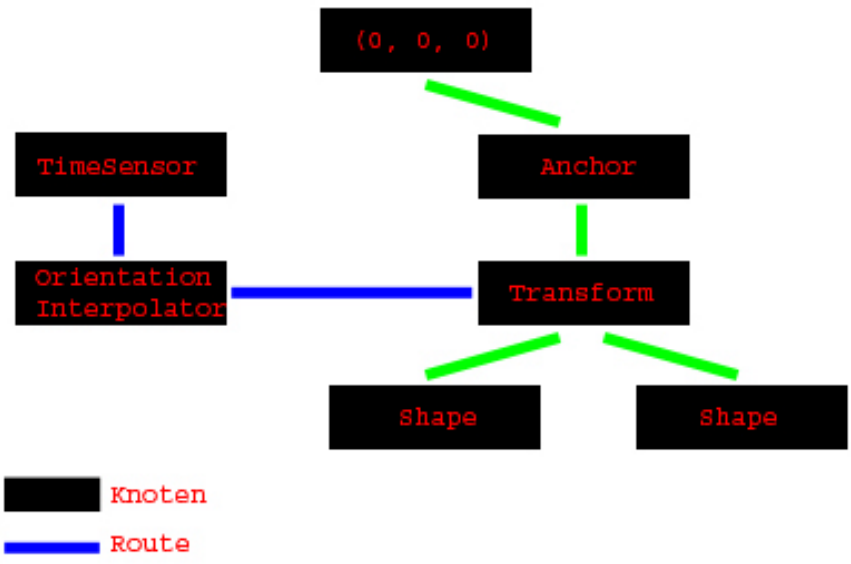

Abbildung 4.1: Einfacher VRML-Szenengraph

[8] Ein Szenengraph ist eine objektorientierte Programmierschnittstelle zur Realisierung interaktiver 3D-Grafik-Anwendungen. Hierbei wird eine Szene abstrakt beschrieben, indem Geometrien, Materialien, Transformationen, Lichtquellen, etc. in einem azyklischen Graphen angeordnet werden.

Bei dem abgebildeten Szenengraph befindet sich unmittelbar unter dem Ursprung (0, 0, 0) ein Anchor-Knoten. Dies entspricht einem Link und somit führt ein Mausklick auf die im Szenengraph folgenden Körper zum Laden der entsprechenden Resource (z. B. eine HTML-Seite) aus dem Netz. Als nächstes folgt ein Transform-Knoten. Dies ist ein erweiterter Gruppenknoten, bei dem auch eine Verschiebung, Rotation oder Skalierung erfolgen kann. Unterhalb dieses Knotens folgen dann noch zwei Shape-Knoten. Wobei einer davon den Text „Hallo Welt“ und der andere einen blau leuchtenden Würfel enthält.

Die Animation erfolgt durch den TimeSensor-Knoten. Dieser sendet in einem bestimmten Zeitabstand ein Time-Event über die Route zum OrientationInterpolator-Knoten, welcher daraufhin ein Rotationsevent erzeugt und an den Transform-Knoten weiterleitet. Da die beiden Shape-Knoten am Transform-Knoten hängen, wirkt sich das auf die beiden aus und es erfolgt eine Drehung.

Folgend noch der VRML-Code der der obigen Szenengraph-Abbildung entspricht.

```
#VRML V2.0 utf8
Background
{
  skyColor 0 0 0.25
}

Anchor
{
  children
  [
    DEF textlink Transform
    {
      children
      [
        Shape
        {
          appearance Appearance
          {
            material Material
            {
              emissiveColor 0 0 1
            }
          }
          geometry Box {}
        }
        Shape
        {
          geometry Text
          {
            string [" Hallo Welt"]
          }
        }
      ]
    }
  ]
  description "FH Köln, Abt. GM"
  url "http://www.gm.fh-koeln.de"
}

DEF rotation
OrientationInterpolator
{
  key [0, 0.5, 1]
  keyValue
  [
    0 1 0 0,
    0 1 0 3.14,
    0 1 0 6.28
  ]
}

DEF uhr TimeSensor
{
  cycleInterval 15
  loop TRUE
}

ROUTE uhr.fraction_changed TO
rotation.set_fraction
ROUTE rotation.value_changed TO
textlink.set_rotation
```

4.4 Persistenz in VRML

Da die Virtual Reality Modeling Language selbst schon als Austauschformat für dreidimensionale Daten und virtuelle Welten verwendet wird, findet sich hier kein eigentliches Verfahren zur Speicherung der Daten zur Laufzeit.

Jedoch ein besonderer Ansatz hierfür besteht in der Verknüpfung von VRML, PHP[9] und einer Datenbank. [VT04] In diesem Fall wird innerhalb der virtuellen Szene der Aufruf einer PHP-Seite ausgelöst. Sei dies nun durch einen Klick auf irgendein dafür gekennzeichnetes Objekt oder durch einen Skriptknoten, der auf ein bestimmtes Ereignis reagiert. Nun werden bei diesem Aufruf der Seite auch alle nötigen Variablen übergeben. Da PHP einen enormen Funktionsumfang bietet, ist es kein nennenswertes Problem die VRML-Werte von dem Skript in eine Datenbank, z. B. MySQL, eintragen zu lassen.

Nun kann man mit PHP auf einfachste Art eine VRML-Welt erzeugen. Z. B. so:

```
<?php
   header ("Content-type: model/vrml");
   echo "#VRML V2.0 utf8\n";
?>
```

Desweiteren ist auch eine Vermischung von PHP und VRML möglich, genauso wie dies auch bei HTML und PHP der Fall ist. Dies sähe dann z. B. so aus:

```
<?php
   header ("Content-type: model/vrml");
   echo "#VRML V2.0 utf8\n";
?>

Shape
{
  geometry Text
  {

<?php
   $text = “Hello World”;
   echo "    string[“.$text.”]\n”;
?>

   }
}
```

[9] PHP (Akronym für "PHP: Hypertext Preprocessor") ist eine weit verbreitete und für den allgemeinen Gebrauch bestimmte Open Source Skriptsprache, welche speziell für die Webprogrammierung geeignet ist, und in HTML eingebettet werden kann.

Der ausgegebene Text, der in der Variable *$text* steht, hätte genauso gut als Argument über die URL beim Aufruf der PHP-Seite übergeben werden können. Beispielsweise: *http://www.irgendeine-url.de/test.php?text=Hello%20World*

Erweitert betrachtet, könnte auch das verwendete Geometrie-Objekt variabel definiert werden usw. Allerdings ist die Länge einer URL beschränkt und ihr maximales Ausmaß abhängig von Browser, Browserversion und Webserver. Somit eignet sich diese Form der Übergabe auf gar keinen Fall für komplexe Welten.

Aber um wieder auf die zuvor erwähnte Datenbank zurückzukommen. Ein entsprechendes PHP-Skript könnte die einzelnen Datensätze auslesen und daraus wieder, auf die oben beschriebene Art und ohne Beschränkungen, eine VRML-Welt generieren.

4.5 eXtensible 3D (X3D)

X3D stellt die bisher letzte Modifikation des VRML 2.0 - Standards dar. Grundlegende Prinzipien wie die Beschreibung einer Szene als hierarchischer Szenengraph, der aus verschiedenen Knoten besteht, oder die Methoden der Animation und Interaktion bleiben dabei erhalten. Allerdings wird die alte VRML 97 - Syntax abgelöst und durch XML ersetzt. [Eib01]

So würde z. B.:

```
Shape
{
  appearance Appearance
  {
    material Material
    {
      emissiveColor 0 0 1
    }
  }
  geometry Box {}
}
```

in X3D wie folgt aussehen:

```
<Shape>
  <Appearance>
    <Material diffuseColor="0.0 0.0 1.0"/>
  </Appearance>
  <Box/>
</Shape>
```

Allerdings ist X3D abwärtskompatibel zu VRML 2.0. Mithilfe von entsprechenden XSL-Stylesheets lassen sich die X3D-Dateien problemlos zurückkonvertieren und in den jeweiligen Browsern oder mit Browser-Plugins betrachten. [Eib01]

X3D ist im Gegensatz zu VRML modular aufgebaut. So ist es vorgesehen, das X3D über eine kleine Menge von unabdingbaren Elementen wie Geometrie, Animation und Beleuchtung verfügt, welche zugleich den Kern dieses Konzepts bilden. [Eib01] Ein Element ist hierbei ein Knoten mit seiner dazugehörigen Funktionalität. Neben diesem immer notwendigen Kern können weitere Module, mit individueller Funktionalität und Schnittstellen, hinzugefügt werden um die speziellen Bedürfnisse der jeweiligen Anwender zufrieden zu stellen.

Durch den modularen Aufbau und die Erweiterbarkeit, bietet sich auch die Möglichkeit in Größe und Laufzeitverhalten optimierte X3D-Browser zu entwickeln, die neben dem Kernmodul nur noch das anwendungsspezifische Modul berücksichtigen.

4.6 Fazit

Da VRML im Gegensatz zu dem später in Kapitel 6 beschriebenem Avango auf „Open Inventor“ anstatt auf „OpenGL Performer“ basiert, ergeben sich zwangsläufig Unterschiede in den verschiedenen Möglichkeiten.

Open Inventor ist beispielsweise darauf ausgelegt, die Entwicklung von 3D-Anwendungen auf herkömmlichen Desktop-Systemen zu vereinfachen. Angehende Grafikprogrammierer die Schwierigkeiten mit der „low-level“ Programmierung in OpenGL haben, haben mit Open Inventor ein Werkzeug zur Verfügung, das die Anwendungsentwicklung auf einem höheren Level ermöglicht. [SGI04]

OpenGL Performer hingegen ist darauf optimiert die bestmögliche Performanz innerhalb der Anwendungen zu erzielen. Vorzugsweise für Echtzeit-Simulationen, Virtual Reality oder ähnliche Einsatzgebiete die eine hohe Performanz voraussetzen. [SGI04]

Weitere Unterschiede ergeben sich natürlich durch die Modifikationen und Erweiterungen durch VRML bzw. Avango. Die Entwicklung in Avango findet beispielsweise auch auf einem höheren Level als die in VRML statt. So entstehen letztendlich unterschiedliche Möglichkeiten bei der Beschreibung einer Szene. Dadurch eignet sich VRML bzw. X3D nicht direkt als Datenformat für Avango. Zwar bietet X3D durch den modularen Aufbau Spielraum für nötige Erweiterungen, allerdings wären Teile des immer vorhandenen X3D-Kerns überflüssig. Außerdem bestünde eine gewisse Abhängigkeit, die, im Gegensatz zu einem proprietären Format, eventuelle Anpassungen innerhalb des Software-Frameworks Avango notwendig machen würde.

Mit einem eigenen Format könnte die anwendungsspezifische, interne Repräsentation des Szenengraphen in einer beliebigen Granularität serialisiert werden. Sofern dieses neue Format auf XML basieren würde, also es ein XML-Dialekt wäre, entstünden, dank vorhandener Mechanismen zur Umwandlung von XML-Dateien, keinerlei Kompatibilitätseinbußen. Aber mehr zu der eXtensible Markup Language im nächsten Kapitel.

Kapitel 5

eXtensible Markup Language (XML)

“XML will be to data portability what Java is to application portability.”

[Lea99]

Die eXtensible Markup Language ist eine Technologie, die es erlaubt eine unbegrenzte Zahl an Auszeichnungssprachen (engl. Markup Languages) zu definieren. Diese neuen Sprachen können den unterschiedlichsten Zwecken dienen, aber trotzdem auf eine standardisierte Art und Weise verwendet werden. [Bos98]

5.1 Beschreibung

XML wurde bei seiner Einführung 1998 überschwänglich von der Fachpresse gefeiert. Teilweise auch zu Recht, da XML enormes Potential hat. Und getragen von der Euphorie-Welle entstanden schon viele Neuentwicklungen auf der Basis von XML, bzw. Änderungen älterer Datenformate zu XML. Somit hat sich die eXtensible Markup Language inzwischen schon vielerorts als Standard für eine einheitliche Datenspeicherung etabliert. Allerdings ist XML eigentlich gar nichts Neues, sondern eine Ableitung der *Standard Generalized Markup Language (SGML)*, welche bereits seit 1986 international standardisiert ist. [SH01] Die SGML ist auch keineswegs veraltet oder schlechter im Vergleich zu XML. Ganz im Gegenteil, die Sprache ist in ihrem Umfang zu komplex. Nicht nur für „normale“ Anwender die höchstens einen Teil der SGML-Funktionen gebrauchen könnten, sondern auch übliche Webbrowser wären damit überfordert. Also hat man den Funktionsumfang einfach eingeschränkt und das Ganze dann als XML bezeichnet. [BT99]

Somit ist XML, zwangsläufig genau wie SGML, eine Sprache zur Definition von Auszeichnungssprachen. Es dient also sozusagen für die Entwicklung neuer Datenformate. Somit dürfte der Großteil der Anwender nur am Rande mit XML in Berührung kommen und zwar bei der Verwendung eines neu definierten Datenformats. Das eigentliche Format hingegen, wird naturgemäß von einer Minderheit festgelegt, weil sich einfach weniger Leute für diese Arbeit interessieren und sich damit beschäftigen. [SH01]

Auszeichnungssprachen basieren darauf, Informationen von ihrer Darstellungsform zu trennen. Betrachtet man nun ein beliebiges Dokument, so lässt sich dieses in drei Komponenten unterteilen (siehe Abbildung 5.1). Allerdings ist diese Unterscheidung oftmals nur für den menschlichen Betrachter möglich, da Inhalt und formale Layoutanweisungen innerhalb der Datenstruktur vermischt wurden und deshalb für die Maschine nicht trennbar sind. Zumindest ohne weiteren Aufwand.

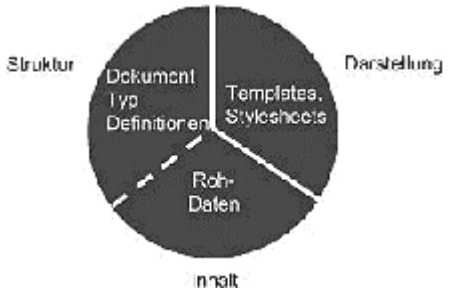

Abbildung 5.1: Dreiteilung von Dokumenten aus [SW00]

Die Komponenten eines Dokumentes sind im Einzelnen [SW00]:

- *Struktur*
  Eine inhaltliche Definition der Einzelinformationen und ihrer Abfolge bzw. Verschachtelung.
- *Darstellung*
  Formale Regeln die festlegen wie die Daten auf einem möglichen Ausgabemedium repräsentiert werden. Stylesheets enthalten Anweisungen zur Formatierung und Positionierung der Informationen.

- *Inhalt*

  Die Informationen die entsprechend der Strukturdefinition in Datenelementen abgebildet werden. Dabei ist es wichtig, dass die Beziehung zwischen Inhalt und den zugeordneten Metainformationen innerhalb der Strukturdefinition erhalten bleibt. Beispielsweise muss die Überschrift einer Nachricht stets mit dem Inhaltselement, welches in der Strukturdefinition die Titelzeile beschreibt, fest verbunden bleiben. Nur so bieten sich sinnvolle Möglichkeiten zur automatisierten Verarbeitung.

Getreu diesem Konzept beinhaltet eine XML-Datei beispielsweise auch nur reine Daten. Diese unterschiedlichen Informationen sind innerhalb der Datei durch sogenannte „Marken“ (engl. Tags) gekennzeichnet und erhalten dadurch eine ihnen fest zugeordnete Bedeutung. Eine dazugehörige Document Type Definition (DTD) bzw. inzwischen eher ein entsprechendes XML-Schema legt die Struktur der Daten fest. D. h. welche Tags erlaubt sind und wie sich diese untereinander verschachteln lassen. Die Darstellung und Verwendung hingegen ist entweder abhängig von der verwendeten Applikation und muss somit auch innerhalb dieser festgelegt werden, oder wird durch die eXtensible Style Language (XSL) beschrieben. XSL bietet Möglichkeiten zur Definition von Regeln nach denen ein XML-Dokument abgearbeitet und dargestellt oder umgewandelt wird.

5.2 eXtensible Style Language (XSL)

Die Sprache XSL setzt sich aus zwei wichtigen Komponenten zusammen [SH01]:

1. aus einer Komponente zur Formatierung von XML-Daten
2. aus einer Komponente zur Transformation von XML-Daten

Die Formatierungskomponente wird auch als XSL-FO (XSL Formatting Objects) bezeichnet und die Transformationskomponente wird allgemein mit XSLT abgekürzt und verwendet obendrein noch eine Hilfssprache namens Xpath, durch die XSLT seine Aufgaben erst wahrnehmen kann.

Xpath hat dabei drei wichtige Aufgaben [SH01]:

1. Adressierung von Daten
2. Definition logischer Ausdrücke
3. Zusätzliche Funktionen

XSL bietet, ähnlich wie CSS[10], eine große Anzahl an verschiedenen Style-Eigenschaften. Darüber hinaus enthält es aber auch Mechanismen, die den logischen Ablauf der Datenpräsentation steuern, wie z. B. automatische Kapitelnummerierung oder Sortierfunktion. Obgleich XSL „nur“ als eine Formatsprache gilt, sind darin durchaus Konstrukte wie bedingungsabhängige Anweisungen oder Schleifen, wie man sie eher aus einer Programmiersprache kennt, enthalten. [SH01]

Zusammen mit der Transformationskomponente XSLT bietet XSL aber noch bedeutend mehr. Damit lassen sich XML-Daten beliebig umwandeln, von einem XML-Dialekt in einen anderen, in eine andere Auszeichnungssprache wie beispielsweise HTML oder aber auch in gänzlich andere Formate wie PDF[11] oder reinen Text.

5.3 Die Vorteile von XML

Anders als in HTML haben Tags bei XML keine vordefinierte Bedeutung. [Bos98] Sie beschreiben auch nicht wie die Daten dargestellt werden sollen, sondern um was für Daten es sich handelt. Dies ist auch ohne weiteres möglich, da die Anzahl der Tags, im Gegensatz zu HTML, nicht beschränkt ist. Mithilfe einer DTD oder eines Schemas lassen sich auf einfache Art und Weise Neue definieren.

Dank dieser „Freiheit“ können Anwender Auszeichnungssprachen entwickeln, die vollkommen auf ihre jeweiligen Anforderungen abgestimmt sind. Und obgleich diese Sprachen dann auch noch so unterschiedlich sein mögen, so können sie doch einheitlich eingelesen und überprüft werden, da sie prinzipiell alle gleich aufgebaut und in Struktur, Darstellung und Informationen getrennt sind.

[10] CSS steht für Cascading Stylesheets und bezeichnet eine HTML-Ergänzungssprache zur Formatierung von HTML-Elementen.
[11] Portable Document Format

Die Darstellungsform wird auch, wie bereits erwähnt, separat angegeben. Dies ist mitnichten ein umständlicher Nachteil, sondern bietet vielmehr die Möglichkeit, auf unterschiedliche Art und Weise auf die Daten zuzugreifen. Sozusagen verschiedene Sichten auf dieselben Daten, was viele neue Möglichkeiten bietet. So wird in [BT99] Brian Kernighan mit *"The problem with "What You See Is What You Get" is that what you see is all you've got,"* zitiert. Dieses Problem gehört damit der Vergangenheit an.

Nehmen wir beispielsweise einen Versandhandel der auf seiner Homepage über einen Online-Shop verfügt. Verändert man nun Werte wie die Anzahl eines Produktes o. ä., oder wechselt innerhalb des Online-Kataloges die Produktkategorie dann wird zur Darstellung der veränderten Werte wie Gesamtkosten oder für eine Übersicht der neuen Produkte jedes Mal eine Anfrage an den entfernten Server für eine komplette neue Seite gesendet. Und dieser Server könnte mitunter viele Anfragen bekommen und überlastet sein, während der eigene Computer mit nichts beschäftigt ist und nur darauf wartet, eine Antwort von eben jenem Webserver zu bekommen. Wären die Kataloginformationen als XML-Datei mitsamt einem kleinen Java-Programm, das unterschiedliche Regeln für die Darstellung der Daten anbietet, verschickt worden, so könnten die Kunden in kürzester Zeit die Informationen individuell betrachten ohne in irgendeiner Form auf den Server angewiesen zu sein.

Ein weiterer Grund warum XML so kompatibel und damit so softwareunabhängig ist, liegt in der Tatsache, das die Zeichencodierung in Unicode[12] erfolgt. Aus diesem Grund können Anwendungen die in der Lage sind XML zu lesen, auch mit asiatischen, arabischen oder kyrillischen Zeichen umgehen. Wohingegen Software deren genutzte Zeichencodierung auf eine bestimmte Region zugeschnitten ist, dazu nicht in der Lage ist.

[12] Unicode ist ein Verfahren zur Kodierung von Zeichen. Es liefert eine eindeutige Nummer für jedes mögliche Zeichen, egal auf welchem Betriebssystem, welcher Software und welcher Sprache. [Uni04]

5.4 Fazit

Durch den großen Spielraum bei der Modellierung des Datenformats bleiben bei XML die Vorzüge eines gänzlich anderen, proprietären Formats erhalten. D. h. keinerlei Abhängigkeiten zu eventuellen Modifikationen am Format, keine notwendigen Anpassungen innerhalb der eigenen Software und die Möglichkeit zur Entwicklung eines Formats das für die eigenen Anwendungen optimiert ist.

Zeitgleich bietet XML aber auch die Vorteile einer inzwischen etablierten Technologie. Frei verfügbare Software-Bibliotheken vereinfachen und verkürzen die Entwicklungsphase. Außerdem ist, durch die standardisierte Verarbeitungsweise von XML-Daten und die daraus resultierenden Transformationsmechanismen, eine hohe Kompatibilität zu anderen Datenformaten gewährleistet.

Aufgrund dieser zahlreichen Vorteile wurde im Rahmen dieser Arbeit auf einen anwendungsspezifischen XML-Dialekt als Datenformat für die Speicherung und Wiederherstellung eines Avango-Szenengraphen hingearbeitet.

Kapitel 6

Die Software-Schnittstelle

In diesem Kapitel wird auf den praktischen Teil dieser Diplomarbeit eingegangen. Es beinhaltet eine einleitende Beschreibung der Aufgabenstellung, einen Überblick über die verwendete Software, sowie Informationen über die implementierte Software-Schnittstelle und des definierten XML-Schemas.

6.1 Beschreibung

Im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Projekts „ProViT“ soll das VR-Software Framework Avango erweitert werden. Und zwar soll die zugrundeliegende Datenstruktur des Szenengraphen um ein Konzept zur Unterstützung von Persistenz erweitert werden. Da auch externe Anwendungen Zugriff auf die persistenten Daten haben sollen, eignet sich XML, aufgrund der im vorherigen Kapitel bereits erwähnten Vorteile, hervorragend für diese Aufgabe.

6.1.1 Das ProViT - Projekt

Verschiedene geographisch entfernte Standorte werden mithilfe von Hochleistungsnetz-werken so verbunden, das immersive Virtual-Reality-Sitzungen zwischen den unter-schiedlichen Standorten übertragen und somit verteilt werden können. Neben einer 3D-Darstellung von neuen Produkten werden auch Audio- und Videosignale der beteiligten Personen verteilt und es ergeben sich immersive Virtual-Reality-Konferenzen, die mitunter große Entfernungen überbrücken. Standort- und unternehmensübergreifende Design Reviews werden auf einfache Weise ermöglicht und dies bei deutlich geringerem Aufwand. [Pro04]

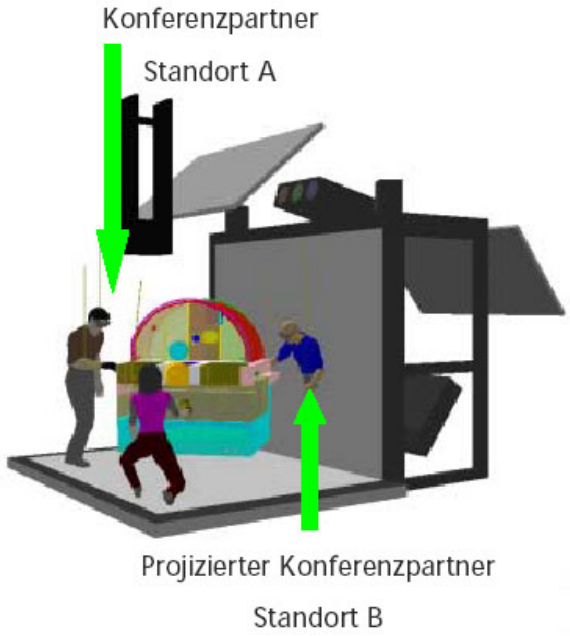

Abbildung 6.1: immersive Virtual-Reality-Konferenz aus [Pro04]

Um diese Zielsetzung zu erreichen bedarf es zunächst einmal eines Systems, welches es erlaubt, innerhalb von virtuellen Umgebungen interaktiv auf einen gemeinsamen Datenbestand zuzugreifen und diesen auch zu visualisieren. Das Virtual-Reality-Software-Framework Avango bietet diese Funktionalität und wird deshalb hier eingesetzt.

Desweiteren bedarf es Komponenten die physikalisch basierte Simulationen und direkte audio-visuelle Kommunikation ermöglichen.

Dieses Projekt ist eine Kooperationsarbeit von mehreren Instituten. Folgende beteiligen sich daran:

- IMK, Fraunhofer Institute for Media Communication, Sankt Augustin
- IPK, Fraunhofer Institute for Production Systems and Design technology, Berlin
- IGD, Fraunhofer Institute for Computer Graphics, Darmstadt
- IPT, Fraunhofer Institute for Production Technology, Aachen
- ZGDV, Computer Graphics Center, Rostock

6.1.2 Avango

Avango ist ein objekt-orientiertes Software-Framework zur Entwicklung von verteilten, interaktiven Virtual-Reality-Anwendungen. Entwickelt wurde und wird Avango in der Abteilung Virtual Environments (VE) des Fraunhofer Institut für Medienkommunikation (ehemals Gesellschaft für Mathematik und Datenverarbeitung (GMD)).

6.1.2.1 Grundlagen

Um die bestmögliche Leistungsfähigkeit für VR-Anwendungen zur Verfügung zu haben, wurde SGI Performer als Basis für Avango gewählt. Dabei kümmert sich Performer um die Kommunikation mit der Graphik-Hardware als auch um aufwendigere Renderaufgaben wie z. B. „culling“. Desweiteren verfügt Avango über eine Sprachanbindung an die Interpreter-Sprache Scheme. Alle Objekte können mithilfe von Scheme erstellt und manipuliert werden. [Tra99]

Die Anwendungsentwicklung lässt sich in zwei Bereiche unterteilen:

1. Existierende Avango-Klassen werden in C++ vererbt und erweitert um komplexe und zeitkritische Funktionalitäten zu realisieren.
2. Die eigentliche Anwendung wird in Form von Scheme - Skripten erstellt. Die einzelnen Skripte erzeugen Objektinstanzen, rufen deren Funktionen auf, manipulieren deren Werte und definieren die Verhältnisse der einzelnen Objekte untereinander. Ein großer Vorteil ist, das diese Scheme - Skripte während der Laufzeit der Anwendung geschrieben, getestet und verbessert werden können. Dadurch verringert sich der zeitliche Aufwand bei der Entwicklung enorm.

Jegliche Objekte werden als FieldContainer bezeichnet. Diese wiederum verfügen je nach Typ über eine unterschiedliche Anzahl an Fields, von denen jedes mindestens einen Wert enthält und deren Summe den Status des Objekts widerspiegelt. Es gibt SingleFields, welche, wie der Name schon andeutet, einen Wert enthalten und MultiFields, welche einen STL[13] - Vektor mit einer beliebigen Anzahl an Werten beinhalten.

[13] STL steht für Standard Template Library

Eine besondere Form der Fields stellen die Adaptor-Fields dar. Jene dienen dazu die Avango API mit der bestehenden Performer API zu verbinden. So lässt sich die Funktionalität der Performer-Knoten in Avango verwenden. Alle für die Szene relevanten FieldContainer werden auch hier innerhalb eines Szenengraphen ausgehend vom Ursprung der Szene angeordnet.

Zwischen Fields, mit einem kompatiblen Datentyp, lassen sich sogenannte Field-Connections errichten. Eine FieldConnection besteht aus einem Quellfeld und einem Zielfeld, wobei das Zielfeld den Wert des Quellfeldes annimmt. Eine Wertänderung im Quellfeld wird unmittelbar über die FieldConnection an das Zielfeld weitergeleitet, ohne den Umweg über die Verbindungen des Szenengraphen verwenden zu müssen. Somit lassen sich zusätzliche Abhängigkeiten oder Beziehungen zwischen den Field-Containern definieren, die allein mit den Möglichkeiten die der Szenengraph bietet, nicht realisierbar wären. Dies unterstützt unter anderem auch die Verwirklichung von interaktivem Verhalten der Anwendung.

Eine weitere Komponente von Avango stellen die Sensoren dar. Sie bilden die Schnittstelle zur realen Welt. Technisch werden Sensor-Klassen nur von Avango’s Field-Containern aber von keiner Performer-Klasse abgeleitet. Dieser Umstand führt dazu, das sie nicht im Szenengraph, der ja letztendlich auch nur auf SGI Performer basiert, auftauchen oder gar eingefügt werden können. Mithilfe von diesen Sensoren kann auf die generierten Daten der unterschiedlichsten Eingabegeräte zugegriffen werden. Sobald ein Gerät neue Werte liefert, werden die entsprechenden Fields des Sensors aktualisiert. Im Zusammenspiel mit FieldConnections zu FieldContainern innerhalb des Szenengraphen besteht die Möglichkeit zur Interaktion mit der Anwendung. Daraus ergibt sich ein orthogonal zum Szenengraph verlaufender Datenflussgraph. [Tra99]

6.1.2.2 Verteilung

Zeitkritisches Rendern einer Szene erfordert schnellen Zugriff auf die Werte der Objekte des Szenengraphen. Gäbe es nur einen Szenengraphen auf einem Server innerhalb der verteilten Anwendung, so wäre die Bandbreite für die Übertragung der Daten zu allen anderen beteiligten Prozessen zu gering. Deshalb ist es notwendig, das

jeder Prozess über eine eigene lokale Kopie des Szenengraphen verfügt. Desweiteren müssen diese Kopien ständig synchronisiert werden. Um dies zu ermöglichen, wird das Konzept des *„Distributed Shared Memory“* (DSM) verwendet. Während *„Shared Memory“* ein Speichersegment beschreibt, das gleichzeitig von mehreren Prozessen von einer Maschine manipuliert werden kann, meint das DSM-Model ein Speichersegment, das auch von verteilten Prozessen innerhalb eines Netzwerks verwendet werden kann. [Tra99]

Avango-Prozesse können sich selbst mit sogenannten *„Distribution Groups“* verbinden und erhalten nach ihrem Beitritt Zugriff auf das DSM-Segment der besagten Gruppe und eine vollständige Kopie des aktuellen Anwendungsstatus. Durch Letzteres werden auch Prozesse, die zu einem späteren Zeitpunkt beitreten, auf den aktuellen Stand gebracht. Soll nun ein verteiltes Objekt erzeugt werden, so erstellt der jeweilige Prozess zunächst ein lokales Objekt und fügt es daraufhin dem Speichersegment der Gruppe hinzu. Gleichzeitig erfolgt eine Benachrichtigung über eine Veränderung an alle beteiligten Prozesse, so dass diese ihre Kopie an die Neuerung anpassen. Dies ist so einfach möglich, da alle FieldContainer und Fields über eine Streaming - Schnittstelle verfügen, so dass jegliche Informationen in einen Stream geschrieben und aus diesem auch wieder heraus gelesen werden können. Neben verteilten Objekten, können auch noch lokale erzeugt werden, welche nicht im Netzwerk verbreitet werden. Abbildung 6.2 zeigt lokale Objekte der einzelnen Prozesse und verteilte Objekte innerhalb einer „Distribution Group“.

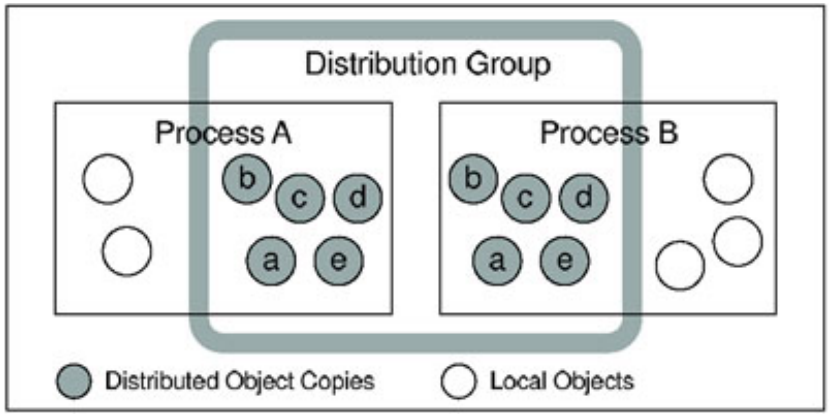

Abbildung 6.2: lokale und verteilte Objekte aus [Tra99]

Bei der Verteilung bilden FieldConnections eine Ausnahme, da sie nur in dem Prozess existieren, der sie auch angelegt hat. Sie werden also nicht verteilt. Dies ist aufgrund der Benachrichtigung von Änderungen und automatischen Aktualisierungen auch nicht nötig, wie auch Abbildung 6.3 verdeutlicht.

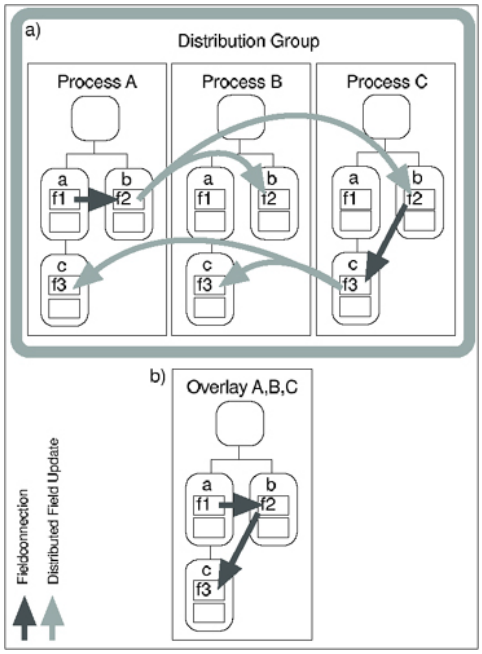

Abbildung 6.3: Gleiches Verhalten bei verteilten und nicht-verteilten Datenflussgraphen aus [Tra99]

Im oberen Teil der Abbildung kann man sehen, wie eine Wertänderung des Fields *f1* sich dank der Verbindung in Prozess *A* auch auf *f2* auswirkt. Daraufhin erfolgt eine Aktualisierung von *f2* innerhalb der anderen Prozesse, wobei die Verbindung in *C* wiederum *f3* manipuliert und zu einer weiteren Aktualisierung der restlichen Prozesse führt. Somit entspricht das Verhalten des verteilten Datenflussgraphen (a) dem eines Eigenständigen (b).

Durch das hier beschriebene System wird die Arbeit von Anwendungsentwicklern deutlich vereinfacht. Sie brauchen sich nicht mit der Umsetzung der Verteilung zu beschäftigen, oder sich gar erst dafür nötiges Wissen anzueignen. Stattdessen können Entwickler wie gewohnt arbeiten, ohne sich anpassen zu müssen. Auch beinhaltet das System einen Szenengraphen, auf den gemeinsam zugegriffen werden kann von allen beteiligten Seiten, wie es z. B. auch schon im zuvor erwähnten Projekt ProViT gefordert wird.

6.1.3 Motivation

Um die Möglichkeiten der VR-Technologie völlig ausschöpfen zu können, ist es nötig über hervorragende Kenntnisse über die Technik zu verfügen. Dies steht allerdings leider oftmals im Widerspruch zu künstlerischen Fähigkeiten. Um nun aber doch mehr Menschen zu solchen Leistungen zu befähigen, ist es nötig diesen Entwicklungsprozess zu vereinfachen.

Vor allem in der Industrie werden Möglichkeiten gebraucht, die die Entwicklungsabteilungen von Firmen in die Lage versetzen, ihre Produkte ohne großen Mehraufwand zu visualisieren. Eventuell für komplexe Simulationen, oder aber auch für Demonstrationen vor Kunden. Das Ganze möglicherweise, wie z. B. im ProViT - Projekt beschrieben, über eine geographische Distanz hinweg. Und auch diese potentiellen Kunden sollten z. B. die Möglichkeit haben, die Angebote ihren Vorstellungen entsprechend zu korrigieren. Oder aber auch verschiedene Abteilungen oder Firmen, die zusammen an einem Projekt arbeiten, sollten sich auf ihre eigentliche Aufgabe konzentrieren können, anstatt sich mit der Realisierung der virtuellen Welt beschäftigen zu müssen.

Dies sind Gründe, wegen derer das objekt-orientierte Virtual-Reality-Software-Framework Avango, wie schon eingangs in diesem Kapitel erwähnt, um ein Konzept zur Unterstützung von Persistenz erweitert werden soll. Die dabei gesicherten Daten könnten dann auf einfache Art und Weise bearbeitet und manipuliert werden.

Allerdings soll dies geschehen, ohne die bestehende Software-Umgebung gravierend zu verändern. Auch sollte das verwendete Datenformat keine größeren Anpassungen nötig machen. Gegen Ende von Kapitel 5 wurde festgestellt, das ein eigener XML-Dialekt für diese Anforderungen die beste Lösung darstellt.

6.2 Verwendete Software-Bibliotheken

Die Programmierung erfolgte in C++ unter Verwendung der ANSI C++ Bibliothek. Um XML Daten lesen und schreiben zu können, wurde die Xerces C++ Bibliothek in der Version 2.4.0 verwendet.

6.2.1 Xerces C++

Diese Bibliothek wird von Apache als Open-Source-Projekt entwickelt und ist folglich kostenlos verfügbar. Um Xerces zwischen unterschiedlichen Betriebssystemen und Versionen kompatibel zu halten, wird die Verwendung von Templates, RTTI[14] und *#ifdef's* innerhalb der Xerces-Bibliothek weitestgehend minimiert. [Apa04]

Einmal eingebunden bietet die Bibliothek XML-Parser und einfache Funktionen zum Erzeugen und Schreiben von XML-Daten. Was bei einer Programmiersprache als Interpreter bezeichnet wird, heißt bei beschreibenden Sprachen wie XML Parser. D. h. in diesem Fall also, das der Parser die XML-Daten Stück für Stück einliest, analysiert und, je nach Option, überprüft, bevor er die Daten der zugrunde liegenden Software zur Verfügung stellt. [SH01]

Zum gegebenen Zeitpunkt waren Level 1, Levl 2 und teilweise Level 3 des Document Object Model (DOM) und die Simple API for XML (SAX) in den Versionen 1.0 und 2.0 innerhalb der Xerces-Bibliothek implementiert.

Das DOM ist eine vom World Wide Web Consortium[15] festgelegte abstrakte Schnittstellenbeschreibung. Hierbei wird ein Dokument (DOMDocument) definiert, das, unabhängig von der jeweiligen Implementation und internen Datenstruktur, nach außen hin objektorientiert repräsentiert wird. Die Objekte des Dokuments werden als Knoten (DOMNode) bezeichnet und innerhalb einer hierarchischen Baumstruktur (DOMTree) angeordnet. Desweiteren definiert das DOM Funktionen zum erzeugen, löschen, durchsuchen, zugreifen und ändern von Dokumentinhalten. [BO00]

SAX hingegen wurde ursprünglich für Java entwickelt und es besteht kein einheitlicher Standard für Umsetzungen in C++, wodurch es logischerweise zu Problemen mit unterschiedlichen Parsern kommen kann. Im Gegensatz zum DOM ermöglicht SAX keinen Zugriff auf die Dokumentinhalte in Form einer Baumstruktur, sondern in Form einer Sequenz von Ereignissen. [Idr99] Diese werden beim einmaligen Durchlaufen eines XML-Dokuments erzeugt und müssen dann von der dahinterliegenden Anwen-

[14] Runtime Type Information, bietet Möglichkeiten zur Laufzeit den dynamischen Typ eines Objektes zu ermitteln.
[15] http://www.w3c.org

dung dazu verwendet werden, ein eigenes Objekt-Model zu erzeugen. Dadurch ist SAX an sich sehr schnell, allerdings bietet es keinerlei Möglichkeiten um XML Dokumente zu erstellen, zu modifizieren oder zu speichern. Auch wahlfreier Zugriff auf Dokumentelemente ist nicht möglich.

In Anbetracht der doch großen Unterschiede zwischen DOM und SAX und der Anforderungen an eine Schnittstelle zur Serialisierung und Deserialisierung, wird hier das DOM verwendet, da es die nötigen Grundfunktionalitäten bietet.

6.3 Implementierung

Wie in Abschnitt 6.1.2.2 ja schon erwähnt, verwendet Avango ein verteiltes und geteiltes Speichersegment worin sich der Szenengraph befindet. Desweiteren gibt es für die unterschiedlichsten Aufgaben (z. B. Zugriff auf die einzelnen FieldContainer usw.) zentrale Dienste mit exklusivem Datenzugriff. Diese Services befinden sich auch in besagtem Speichersegment und sind somit auch von allen Prozessen verwendbar. Programmiertechnisch werden sie alle von der Klasse *fpService* abgeleitet.

Also bietet es sich an, die Persistenz-Funktionalität auch als solch einen zentralen Service zu realisieren. Folglich lege ich eine neue Klasse namens *fpPersistenceService* an welche von *fpService* abgeleitet ist.

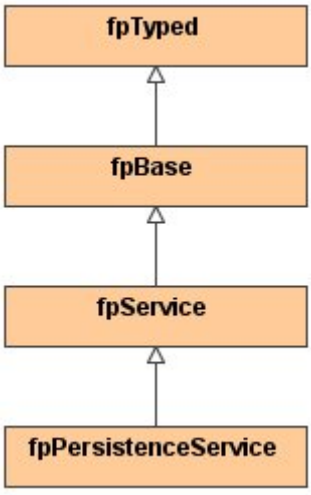

Abbildung 6.4: Vererbungsdiagramm der Klasse fpPersistenceService

Wie die Abbildung 6.4 zeigt ist *fpService* selbst von *fpBase* abgeleitet, welche wiederum ihrerseits von *fpTyped* abgeleitet wurde. Dabei ist Letztere die abstrakte Basisklasse für jegliche Klassen die über Typinformationen zur Laufzeit verfügen sollen. Sie bietet verschiedene Makros an, die von abgeleiteten Klassen verwendet werden müssen, sofern man deren Typ zur Laufzeit erkennen können soll.

Zum weiteren Vorgehen bedarf es allerdings einiger Änderungen, die in den nächsten beiden Abschnitten beschrieben werden.

6.3.1 Felder und deren unterschiedlicher Inhalt

Es ist erforderlich, das alle für die virtuelle Szene relevanten FieldContainer, bzw. deren einzelnen Felder, völlig unabhängig von den darin enthaltenen Daten und deren Typ, ein- und ausgelesen werden können. Wie schon im Abschnitt über die Verteilung innerhalb von Avango (6.1.2.2) erwähnt, ist in jeden FieldContainer und in jedes Field eine Streaming-Schnittstelle implementiert.

Programmiertechnisch sieht das folgendermaßen aus. *fpField* ist auch von *fpTyped* abgeleitet. Außerdem stellt die Klasse die Basis für *fpSingleField* und *fpMultiField* dar (vgl. Abbildung 6.5). Sie enthält folgende virtuellen Funktionen:

read(fpInputStream& is)
write(fpOutputStream& os)

Alle möglichen Felder werden dann entweder als SingleField- oder als MultiField-Template erstellt und verfügen über eine Implementierung der beiden Funktionen. Mithilfe von entsprechenden Ein- und Ausgabeoperatoren für jeden Datentyp können so also alle Felder ihren Wert in einen Stream übertragen und auch wieder daraus herstellen.

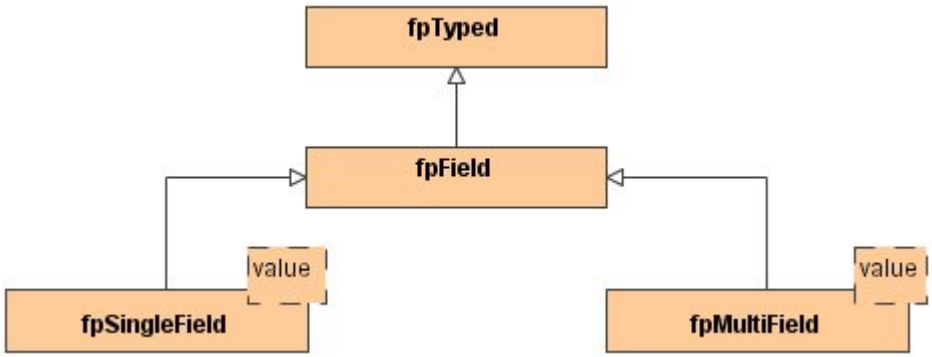

Abbildung 6.5: Vererbungsdiagramm der Klasse fpField

Nun ist dies aber für den hier geplanten Verwendungszweck unzureichend, da mit einem Field (bspw. *Parents* oder *Children*) das einen, oder auch mehrere, Link(s) zu einem FieldContainer enthält, anders verfahren werden muss als mit Einem das „normale“ Werte enthält. So ist es beim Speichern wichtig, das ein Feld mit solch einem Link nicht einfach nur den Wert (in diesem Fall die Adresse des Zielobjektes) in den Stream schreibt und sich dann nicht mehr weiter darum kümmert, sondern das stattdessen noch dafür gesorgt wird, das sich dieser neue FieldContainer eventuell auch persistent macht. Und auch nur „eventuell“, weil ja die Möglichkeit besteht, dass das betreffende Objekt zuvor bereits aufgrund einer anderen Verbindung gesichert wurde.

Dies ist besonders bei der in Kapitel 3 erwähnten transitiven Persistenz, die besagt das alle von einem persistenten Objekt aus erreichbaren Objekte mitgesichert werden müssen, von Bedeutung. Schließlich macht auch genau dieses Prinzip bei einem Szenengraphen Sinn. Soll beispielsweise ein Knoten aus dieser Baumstruktur gespeichert werden, so ist es erforderlich, auch die darunter hängenden Kinderknoten mitzuspeichern. Selbiges trifft auf FieldConnections zu. Ansonsten würden Teile der Szene schlicht und ergreifend fehlen, was folglich eine korrekte Rekonstruktion der Szene unmöglich machen würde.

Beim Wiederherstellen der Daten tritt diese Notwendigkeit natürlich auch ans Tageslicht. Enthält ein eingelesener FieldContainer *A* einen Link zu einem weiteren FieldContainer *B* der noch nicht wiederhergestellt wurde, so ist es erforderlich dafür zu sorgen, das der Link innerhalb von *A* auf ein gültiges *B* verweist. Aber ohne einen existierenden FieldContainer *B* geht das schlecht, da weder eine Speicheradresse noch eine ID vorhanden sind. Um dies zu erreichen, muss also *B* zunächst einmal erzeugt

werden, bevor irgendwelche anderen Objekte in irgendeiner Form darauf verweisen können.

Um nun zur Laufzeit zwischen einem normalen Feld und einem Feld mit einem Link unterscheiden zu können, wurden zwei neue Klassen eingeführt die von *fpField* abgeleitet werden. Namentlich sind das *fpSingleFieldBase* und *fpMultiFieldBase*. Sie bilden die Grundlage für die Templates *fpSFAnyLink* und *fpMFAnyLink*. Gleichzeitig wurde der restliche Quellcode von Avango an diese Neuerung angepasst.

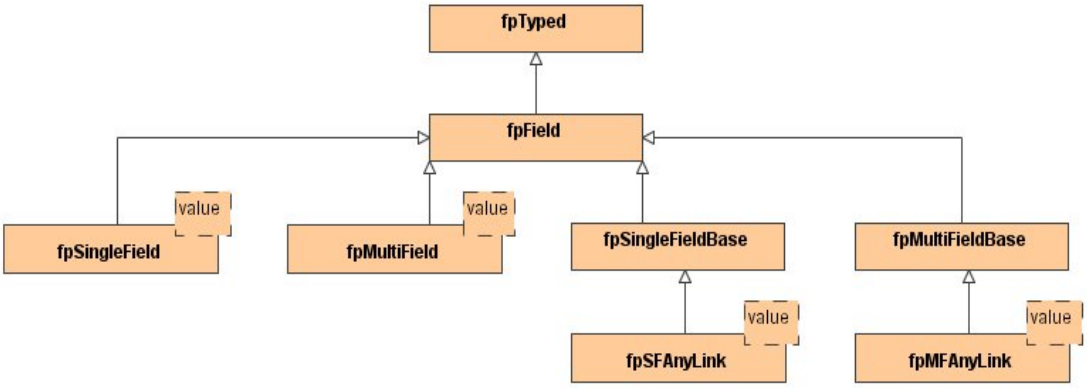

Abbildung 6.6: Vererbungsdiagramm der Klasse fpField nach den Änderungen

Nach dieser Änderung ist es nun möglich, mittels eines *dynamic_cast()*, während der Laufzeit herauszufinden, ob es sich bei dem aktuellen Feldinhalt um einen Link handelt oder nicht.

6.3.2 Objektidentitäten

Als nächstes bedarf es Änderungen bei den Daten, die von einem Link-Feld mittels *write()* in den Stream geschrieben werden. Die zuvor erwähnten Funktionen *read()* und *write()* dienen ursprünglich der Verteilung und lesen und schreiben nur die Speicheradresse des verlinkten Objektes aus dem Stream, bzw. in den Stream. Da alle beteiligten Prozesse das selbe verteilte Speichersegment verwenden, ist dies auch völlig korrekt. Die Speicheradressen sind ja somit logischerweise überall identisch. Allerdings nützen eben diese Adressen nichts mehr, sobald die Anwendung beendet wird. Bei einem Neustart wird ein neuer Adressraum verwendet, sprich ein neues Speichersegment, und alle alten Adressen sind auf einen Schlag nutzlos. Der Fall das genau das identische

Segment wie zuvor von der Anwendung belegt wird ist mehr als unrealistisch und deshalb keine vernünftige Grundlage.

Also ist es für eine korrekte Speicherung der Szene notwendig, alternative Informationen anstatt der Speicheradresse zu sichern. Außerdem, wie schon in Kapitel 3 zu lesen war, treten zum Beispiel bei der „persistence by copying“-Technik Probleme durch das Fehlen von Objektidentitäten auf. Dadurch können Objekte mehrmals erzeugt werden, selbst wenn sie bereits erzeugt wurden. Letztendlich führt dies zu vielen überflüssigen Instanzen und dem Verlust der ursprünglichen Verhältnisse zwischen den „korrekten“ Instanzen.

Nun, FieldContainer in Avango verfügen über eine ID. Setzt man nun voraus, das diese ID auch stets mitgespeichert wird, so eignen sich die IDs der verlinkten FieldContainer wunderbar als Alternative zur Speicheradresse. Auch das Problem mit überflüssigen Instanzen tritt nicht mehr auf, da nun auf einfache Art und Weise überprüft werden kann, ob Objekte bereits wiederhergestellt wurden oder nicht.

Zum gegebenen Zeitpunkt existierten solche IDs zwar, allerdings bestanden jene aus nichts weiter als einem ganzzahligen Wert der bei jedem Neustart auf einen Startwert gesetzt und für jeden FieldContainer einfach um 1 erhöht wurde. Somit waren diese IDs nur innerhalb einer laufenden Anwendung einzigartig. Dieser Umstand würde unweigerlich zu Komplikationen bei der Wiederherstellung der Szene führen, da unterschiedliche Objekte, die in voneinander unabhängigen Programmausführungen erzeugt worden wären, über die gleiche ID verfügen würden.

Auf der anderen Seite existierte allerdings bereits eine Klasse namens *fpUUID* zur Erzeugung von UUIDs (Universal Unique Identifier) die nur noch darauf wartete eingebunden und verwendet zu werden. Unter einer UUID versteht man einen 128-Bit Schlüssel der eindeutig in Raum und Zeit ist. Soll heißen, ein Rechner wird niemals zweimal dieselbe UUID erzeugen und zwei unterschiedliche Rechner werden niemals eine identische UUID erzeugen. Und zwar ganz einfach weil sich diese 128-Bit-Nummern aus MAC-Adresse[16], der aktuellen Zeit und zufälligen Daten zusammensetzen. [UIG04] Aufgrund dieser Zusammensetzung können also nur unterschiedliche

[16] Die MAC-Adresse (Media Access Control) ist eine eindeutige Hardware-Adresse eines jeden Netzwerkgerätes und wird zur Identifikation innerhalb eines Netzwerkes verwendet.

IDs, vorausgesetzt niemand stellt die Systemzeit zurück, erzeugt und vergeben werden. Es besteht somit auch keine Notwendigkeit für die beteiligten Prozesse ihre erzeugten IDs auf Eindeutigkeit innerhalb der verteilten Anwendung, mit den anderen Prozessen, zu überprüfen und eventuell abzugleichen.

Aber kommen wir nun wieder auf die für die Persistenz nutzlosen Speicheradressen und die sinnvollere Speicherung der Objekt-ID zurück. Um dies zu realisieren wurden die Klassen *fpSingleFieldBase* und *fpMultiFieldBase* um zwei Funktionen bereichert. Diese lauten *addValue(fpFieldContainer* fc)* und *writeValue(fpOutputStream& os)*.

Deren Implementierung in den Templates *fpSFAnyLink* und *fpMFAnyLink* sieht wie folgt aus:

fpSFAnyLink:

```
virtual void addValue(fpFieldContainer* fc)
{
  _value.setBasePtr(fc);
  fieldChanged();
}

virtual void writeValue(fpOutputStream& os)
{
  getValue();
  if (_value.isValid())
  {
    fpFieldContainer *fcp = (fpFieldContainer*)_value.getBasePtr();
    os << fcp->getId();
  }
}
```

fpMFAnyLink:

```
virtual void addValue(fpFieldContainer* fc)
{
  Value tmp;
  tmp.setBasePtr(fc);

  add1Value(tmp);
  fieldChanged();
}
```

```
virtual void writeValue(fpOutputStream& os)
{
  getValue();

  if (os.isBinaryEnabled())
  {
    os << _value.size();

    typename vector<Value>::const_iterator i = _value.begin();

    while (i != _value.end())
    {
      if ((*i).isValid())
      {
        fpFieldContainer *fcp = (fpFieldContainer*)(*i).getBasePtr();
        os << fcp->getId();
      }
      i++;
    }
  }
  else
  {
    os << _value.size();
    (ostream&) os << ' ';

    typename vector<Value>::const_iterator i = _value.begin();

    while (i != _value.end())
    {
      if ((*i).isValid())
      {
        fpFieldContainer *fcp = (fpFieldContainer*)(*i).getBasePtr();
        os << fcp->getId();
        (ostream&) os << ' ';
      }
      i++;
    }
  }
}
```

Die Funktion *writeValue()* macht eigentlich nichts anderes, als die UUID des verlinkten Objektes zu ermitteln und diese, anstatt der Adresse, in den Stream zu schreiben. Wohingegen *addValue()* einen FieldContainer übergeben bekommt, dessen aktuelle Speicheradresse ermittelt und diese daraufhin in das entsprechende Feld einträgt.

6.3.3 Die Klasse fpPersistenceService

Nach all diesen Änderungen sollten eigentlich alle Voraussetzungen für die Klasse *fpPersistenceService* gegeben sein. Abbildung 6.7 zeigt den Aufbau der Klasse. Anschließend folgt eine Beschreibung der Klassenattribute und Funktionen.

Abbildung 6.7: Die Klasse fpPersistenceService

6.3.3.1 Beschreibung der Klassenattribute

Alle Attribute dieser Klasse sind privat.

*fpNodeQueryService *nqs;*
Dies ist ein Zeiger auf einen *fpNodeQueryService*, welcher es erlaubt auf die unterschiedlichen FieldContainer und deren Fields zuzugreifen.

typedef std::map<fpUUID, fpUUID> id_map;
id_map idm;
id_map::iterator it;
Da es momentan noch keine Möglichkeit gibt, die ID eines FieldContainers nachträglich zu ändern, bzw. eine vorgegebene UUID bei der Erzeugung zu verwenden, wird diese STL-Map[17] verwendet. In ihr werden ID-Paare gespeichert, die sich aus den beiden ID's eines wiederhergestellten FieldContainers zusammensetzen. Nämlich der ID innerhalb des DOMDocuments und der ID innerhalb des Szenengraphen. Außerdem dient sie zur Überprüfung, ob ein FieldContainer bereits wiederhergestellt wurde, oder eben noch nicht.

bool FirstSaveNode, FirstLoadNode, FirstFragNode;
Bedingt durch einige rekursive Funktionen enthalten diese Variablen Abbruchinformationen die für eben jene Funktionen notwendig sind.

string OutputEncoding, EOLSequence;
bool NormalizeCharacters, SplitCDataSections, SerializationValidation;
bool ExpandEntityReferences, WhitespaceInElemenContent, DiscardDefaultContent;
bool FormatCanonical, FormatPrettyPrint, UseFilter;
Dies sind alles Variablen, die für die Serialisierung von Bedeutung sind. So lässt sich der Xerces-Parser mit ihrer Hilfe konfigurieren bezüglich z. B. Ausgabeformatierung und Überprüfung. Allerdings werden sie bisher alle, bis auf FormatPrettyPrint, im

[17] Standard Template Library - Maps enthalten Wertpaare die in einem Key / Value - Verhältnis zueinander stehen.

Konstruktor der Klasse initialisiert. Aber *configSerialize()* könnte diesbezüglich erweitert werden.

string ValidationScheme;
bool Namespaces, DoSchema, SchemaFullChecking, CreateAllNodes;
Im Gegensatz zu den unmittelbar zuvor behandelten Variablen dienen diese der Konfiguration des Einlesens von XML-Daten. Mit ihnen lässt sich u. a. die Überprüfung mithilfe eines XML-Schemas aktivieren bzw. deaktivieren. Sie werden aber auch alle im Konstruktor gesetzt. Eine eventuell implementierte Funktion *configParse()* könnte dazu dienen, sie zu manipulieren.

bool hasParsedDocument;
Enthält Informationen darüber ob ein eingelesenes DOMDocument existiert.

bool hasCreatedDocument;
Enthält Informationen darüber ob ein DOMDocument angelegt wurde.

DOMDocument doc;*
Ein Zeiger auf das verwendete DOMDocument.

6.3.3.2 Beschreibung der öffentlichen Klassenfunktionen

bool saveSceneGraph(const char XMLFilename);*
Diese Funktion bietet die Schnittstelle zur Speicherung der gesamten Szene. Als Übergabeparameter erwartet sie einen Namen für die XML-Datei die mit den vorhandenen Daten erzeugt werden soll. Sie ruft zunächst die private Funktion *createDOMTree()* im Standardmodus (näheres zu den unterschiedlichen Modi bei der Erläuterung der Funktion *createDOMTree()*) auf um ein DOMDocument mit allen

Szeneninformationen zu erzeugen und anschließend die ebenfalls private Funktion *serialize()*, die, wie der Name schon sagt, zur Serialisierung der Daten dient.

bool saveFragment(const char XMLFilename, fpUUID id);*
Mit dieser Funktion kann nur ein Teil der Szene gespeichert werden. Wie schon bei *saveSceneGraph()* wird auch hier ein Name für die zu erzeugende Datei erwartet und weiterhin die UUID des FieldContainers der den Ursprung des Szenenteils bilden soll. Auch hier wird zunächst die Funktion *createDOMTree()* aufgerufen, allerdings im Modus „fragment“, und anschließend *serialize()*.

*void loadSceneGraph(const char *XMLFilename);*
Hiermit kann eine komplette Szene unter Angabe der XML-Datei geladen werden. Zunächst wird die private Funktion *parse()* aufgerufen, mit der XML-Dateien eingelesen werden können. Danach erfolgt ein Aufruf von *createSceneGraph()* um aus dem zuvor geparsten DOMDocument die FieldContainer der ursprünglichen Szene zu generieren.

*void loadFragment(const char *XMLFilename, fpUUID Parent);*
Analog zu *saveFragment()* existiert auch ein *loadFragment()* für den umgekehrten Fall. Mit dieser Funktion kann ein Szenenteil in die Szene hineingeladen werden. Als Übergabeparameter wird der Name der Datei, als auch die UUID des FieldContainers an den der Szenenteil „angehangen“ werden soll, erwartet. Programmiertechnisch werden die selben Funktionen aufgerufen wie auch schon in *loadSceneGraph()*.

void configSerialize(const bool FPP);
Dies ist die Schnittstelle zur Konfiguration der Serialisierung. Allerdings bisher nur mit der Option den Inhalt der erzeugten XML-Datei in einem „für Menschen gut lesbaren Format“ zu formatieren. Sie bietet also noch reichlich Spielraum für Erweiterungen.

static Elk_Object saveSceneGraphCB(fpTyped obj, Elk_Object);*
Callback-Funktion die es ermöglicht *saveSceneGraph()* aus Scheme heraus aufzurufen.

static Elk_Object saveFragmentCB(fpTyped obj, Elk_Object);*
Callback-Funktion die es ermöglicht *saveFragment()* aus Scheme heraus aufzurufen.

static Elk_Object loadSceneGraphCB(fpTyped obj, Elk_Object);*
Callback-Funktion die es ermöglicht *loadSceneGraph()* aus Scheme heraus aufzurufen.

static Elk_Object loadFragmentCB(fpTyped obj, Elk_Object);*
Callback-Funktion die es ermöglicht *loadFragment()* aus Scheme heraus aufzurufen.

static Elk_Object configSerializeCB(fpTyped obj, Elk_Object);*
Callback-Funktion die es ermöglicht *configSerialize()* aus Scheme heraus aufzurufen.

6.3.3.3 Beschreibung der privaten Klassenfunktionen

bool createDOMTree(const char Mode = "standard", fpUUID ID = fpUUID::null);*
Diese Funktion erstellt einen DOMTree mit allen relevanten FieldContainern. D. h. ganz nach dem in Kapitel 3 aufgeführten Konzept der transitiven Persistenz werden vom „Start-Container" aus, alle weiteren Objekte die in irgendeiner Form (durch FieldConnections oder Links) erreicht werden können, mitverarbeitet und in den DOMTree übernommen. Die zwei optionalen Übergabeparameter dienen der Wahl des angewandten Modus und der Bestimmung des „Start-Containers".

Die Funktion kennt zwei unterschiedliche Modi:

- *standard*
  In diesem Modus wird eine eventuell übergebene ID ignoriert und als Ursprung der FieldContainer mit der Bezeichnung *av-root* verwendet. Dies ist der FieldContainer an dem auch alle relevanten Objekte der virtuellen Szene hängen.
- *fragment*
  Hierbei bezeichnet die übergebene ID den FieldContainer von dem die weitere Verarbeitung ausgeht.

bool createDOMNode(fpFieldContainer fc);*
Eine rekursive Funktion die von *createDOMTree()* aufgerufen wird und einen DOM-Node erzeugt der dem ihr übergebenen FieldContainer entspricht. Allerdings nur, wenn innerhalb des DOMTree's noch kein DOMNode mit der UUID des übergebenen Field-Containers existiert. Trifft die Funktion dabei auf einen Link oder eine FieldConnection zu einem weiteren FieldContainer, so ruft sie sich selbst mit dem neuen Objekt als Parameter auf.

bool createDOMField(DOMNode Parent, fpFieldContainer* fc, const int index) const;*
Diese Funktion wird von *createDOMNode()* verwendet um aus dem Feld an Position *index* innerhalb des FieldContainers *fc* einen entsprechenden DOMNode, mitsamt einem Kindknoten für den Inhalt des Feldes, anzulegen. Der neu erzeugte DOMNode wird letztendlich an den Knoten *Parent*, welcher dem eigentlichen FieldContainer *fc* entspricht, angehängt.

void createSceneGraph(fpUUID Parent = fpUUID::null);
Hiermit werden die im DOMDocument enthaltenen Informationen dem Avango-Szenengraphen hinzugefügt. Wird eine UUID übergeben, so werden die daraufhin neu erzeugten Objekte an den FieldContainer mit dieser ID gehängt, andernfalls, also ohne Angabe von *Parent*, wird die Aktuelle durch eine neu erzeugte Szene ersetzt. Beim Wiederherstellen von Szeneninformationen wird innerhalb der aufgerufenen Funktion

createFieldContainer() darauf geachtet, das „verlinkte“ Objekte zuvor auch erzeugt wurden.

fpFieldContainer createFieldContainer(fpUUID DOMNodeID);*
Hierbei handelt es sich um eine weitere rekursive Funktion. Sie wird von *createSceneGraph()* mit der ID des ersten, wiederherzustellenden Objekts aufgerufen. Zunächst wird überprüft ob sich innerhalb des Szenengraphen schon ein FieldContainer mit der angegebenen ID befindet. Ist dies der Fall, so werden die Feldinformationen dieses FieldContainers aktualisiert. Andernfalls wird gemäß den im DOMNode enthaltenen Typinformationen ein dementsprechender Avango-FieldContainer erzeugt, dessen Felder die zuvor gespeicherten Daten übernehmen. Enthält nun eines dieser Felder einen Link oder eine FieldConnection, also die UUID eines weiteren FieldContainers, so ruft sich die Funktion selbst mit der neuen ID als Parameter auf.

void link(fpSingleFieldBase field, DOMNode* DOMField);*
void link(fpMultiFieldBase field, DOMNode* DOMField);*
Diese überladene Funktion wird von *createFieldContainer()* aufgerufen um die Adresse eines FieldContainers, dessen UUID im Knoten *DOMField* übergeben wird, zu ermitteln und in *field* einzutragen. Je nach Feldtyp (SingleField oder MultiField) wird hierbei entweder nur ein einzelner Link oder ein Vektor von Links ersetzt bzw. manipuliert.

void setFieldValue(fpField field, DOMNode* DOMField) const;*
Diese Funktion wird genauso wie *link()* von createFieldContainer() aufgerufen, allerdings um den Wert eines „normalen“ Feldes *field* an den Inhalt des Knoten *DOMField* anzupassen.

bool addId(DOMElement element, const char* idAttr) const;*
Hilfsfunktion die das Attribut (*idAttr*) welches als ID dienen soll bei einem DOM Knoten (*element*) festlegt und zeitgleich über eine Fehlerabfrage verfügt. Wird bei-

spielsweise nach dem Parsen einer XML-Datei oder dem Erzeugen von bestimmten Knoten verwendet.

bool addChild(DOMNode Parent, DOMNode* Child) const;*
Auch eine Hilfsfunktion mit der sich Kindknoten (*Child*) an Knoten (*Parent*) hängen lassen bei zeitgleicher Fehlerabfrage.

DOMDocument createDocument(const char* Namespace, const char* Name);*
Nach Überprüfung der Variablen *hasParsedDocument* und *hasCreatedDocument* wird ein eventuell im Speicher vorhandenes DOMDocument gelöscht und ein Neues, unter Verwendung von *Name* und *Namespace*, angelegt. Der Rückgabewert besteht aus einem Zeiger auf das neue DOMDocument. Außerdem wird *hasCreatedDocument* gesetzt.

DOMDocument parse(const char* XMLFilename);*
Auch diese Funktion löscht zunächst ein eventuell vorhandenes DOMDocument. Anschließend wird die XML-Datei die durch *XMLFilename* angegeben wurde eingelesen. *hasParsedDocument* wird gesetzt und ein Zeiger auf das gerade eingelesene DOMDocument zurückgegeben.

bool serialize(DOMNode rootNode, const char* XMLFilename) const;*
Diese Funktion dient, wie der Name schon sagt, der Serialisierung eines DOM-Documents. Sofern *hasParsedDocument* oder *hasCreatedDocument* gesetzt sind, werden, ausgehend vom übergebenen *rootNode*, alle untergeordneten Knoten in eine XML-Datei namens *XMLFilename* geschrieben.

6.3.4 das XML-Schema

Nach der Beschreibung der Klasse nun noch eine Übersicht über das hier erstellte Schema. Bei der Definition eines solchen Schemas ist zu beachten, das nur „Elemente“ fähig sind Unterelemente zu haben. Dafür kann allerdings durch Elemente die Menge der erlaubten Werte nicht eingeschränkt werden. Dies ist den „Attributen“ vorbehalten. Auch die Lesbarkeit sollte nicht vernachlässigt werden, weshalb sinnvolle Bezeichner gewählt werden sollten, die etwas über die Art der Daten aussagen.

6.3.4.1 Element: PersistenceService

Zunächst wird ein Element namens *PersistenceService* definiert, welches über ein notwendiges Attribut *start* verfügt. Dieses Attribut beinhaltet die UUID des ersten wiederherzustellenden FieldContainers. Außerdem enthält *PersistenceService* eine beliebig lange Sequenz von *fpFieldContainer-Elementen*. Aus diesem Grund, also weil das Element selbst weitere Elemente enthält, ist es ein sogenannter „complexType“.

```
<xs:element name="PersistenceService">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="fpFieldContainer" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="start" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

6.3.4.2 Element: fpFieldContainer

Als nächstes wird das Element *fpFieldContainer* definiert. Dieses enthält zwei Attribute (*id* und *type*), welche beide benötigt werden. Sie dürfen also nie fehlen, wenn beim Einlesen die XML-Daten anhand des Schemas überprüft werden. Innerhalb von Avango werden die Typ-Informationen genutzt um entsprechende FieldContainer zu erstellen. Ansonsten befindet sich auch hier wieder eine Sequenz, aber diesmal mit einer beliebigen Anzahl an *fpField-Elementen.*

```
<xs:element name="fpFieldContainer">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="fpField" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:integer" use="required"/>
    <xs:attribute name="type" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>
```

6.3.4.3 Element: fpField

fpField wird auch gleich als Nächstes festgelegt. Wie bei den Beiden zuvor, handelt es sich auch hierbei um einen complexType. Das einzige Attribut innerhalb von *fpField* lautet *name* und ist zwingend notwendig. Zusätzlich enthält fpField noch ein „Typ“-Element aus einer Liste von verschiedenen Möglichkeiten, welche innerhalb des *<xs:choice>-Tags* stehen. Hier wurden nun nur einige der Möglichkeiten aufgelistet. Außerdem verfügt ein fpField noch über eventuelle *connection-Elemente*.

```
<xs:element name="fpField">
  <xs:complexType>
    <xs:sequence>
      <xs:choice>
        <xs:element ref="fpSFBool"/>
        <xs:element ref="fpSAFInt"/>
        <xs:element ref="fpSAFUInt"/>
        <xs:element ref="fpSFString"/>
        <xs:element ref="fpSAFString"/>
        ...
      </xs:choice>
      <xs:element ref="connection" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

6.3.4.4 „Typ“-Elemente

Da alle Felder in Avango Templates sind, entstehen durch die verschiedenen Datentypen etliche unterschiedliche Kombinationen, bzw. Feldtypen. Diese lassen sich nun auch noch gesondert definieren, damit gewährleistet ist, das der eigentliche Feldinhalt über den richtigen Datentyp verfügt. Nachfolgend nur einige Beispiele.

```
<xs:element name="fpSFBool" type="xs:integer"/>
<xs:element name="fpSAFInt" type="xs:integer"/>
<xs:element name="fpSAFUInt" type="xs:integer"/>
<xs:element name="fpSFString" type="xs:string"/>
<xs:element name="fpSAFString" type="xs:string"/>
...
```

6.3.4.5 Element: connection

Letztendlich wird noch das *connection-Element* definiert. Welches zwei benötigte Attribute beinhaltet. Und zwar *FC_ID* und *Index*, mit deren Hilfe sich FieldConnections innerhalb von Avango wiederherstellen lassen.

```
<xs:element name="connection">
  <xs:complexType>
    <xs:attribute name="FC_ID" type="xs:string" use=”required”/>
    <xs:attribute name="Index" type="xs:integer" use=”required”/>
  </xs:complexType>
</xs:element>
```

6.4 Abschlussbetrachtung

Mit der Klasse fpPersistenceService verfügt das Software-Framework Avango über einen neuen zentralen Service zur Serialisierung und Deserialisierung seines Szenengraphen. Damit wird die Forderung nach einem Konzept zur Unterstützung von Persistenz erfüllt.

Es zeichnet sich sowohl durch die in Kapitel 3 angesprochene Persistenzunabhängigkeit als auch durch die Transitive Persistenz aus. Schließlich macht es für die Anwendung keinen Unterschied, ob sie mit persistenten Daten arbeitet oder nicht. Weiterhin werden bei der Serialisierung eines Objekts, alle potentiell erreichbaren Objekte mitgespeichert.

Allerdings enthält Avango auch Objekte die nicht zum Szenengraphen gehören bzw. die bei Szenenteilen nicht berücksichtigt werden, wodurch sie bei der Speicherung quasi aussortiert werden. Dadurch ist keine Typ Orthogonalität gewährleistet und es handelt sich hierbei somit auch nicht um eine Orthogonale Persistenz.

Zur Realisierung der Software-Schnittstelle wurde Avango nur in unbedingt notwendigen Bereichen erweitert. Beispielsweise um eindeutige Objektidentitäten zu erzeugen und zu verwenden, die auch noch in einer späteren Programmausführung einzigartig sind.

Der definierte XML-Dialekt erlaubt eine 1:1 Repräsentation von Avango-Objekten. Somit sind stets alle Objektinformationen gesichert und es bedarf keiner Modifikationen an der bestehenden Software-Umgebung. Externe Anwendungen mit einer anderen Repräsentation eines Szenengraphen können, dank der standardisierten XML-Technologie, ihre „Sicht“ auf diese Daten ganz an ihre Anforderungen anpassen oder aber auch die benötigten Daten in ihr Format transformieren.

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurden Möglichkeiten zur Realisierung eines Persistenz-Konzepts innerhalb eines Virtual-Reality-Software-Frameworks betrachtet und untersucht. In diesem Zusammenhang wurde eine dementsprechende Software-Schnittstelle beschrieben und im praktischen Teil der Arbeit als Erweiterung der VR-Entwicklungsumgebung Avango implementiert.

In diesem letzten Kapitel, erfolgt eine Zusammenfassung der bisherigen Arbeit und zum Abschluß der Arbeit ein Ausblick auf mögliche Optimierungen bzw. Erweiterungen der Software-Schnittstelle, als auch ein Ansatz für die unterschiedliche Verwertbarkeit des Datenformats.

7.1 Zusammenfassung

Das Interesse an der virtuellen Realität und dementsprechenden Anwendungen steigt seit Jahren. Die Entwicklung wird in den unterschiedlichsten Bereichen vorangetrieben und führt zu ebenso unterschiedlichen Technologien. Das Kapitel *Virtuelle Realität* befasst sich mit dieser Thematik. Darin wird zunächst die Herkunft verschiedener zusammenhängender Begriffe erläutert, bevor eine allgemeingültigere Definition gebildet wird. Danach wird auf die technologischen Voraussetzungen die nötig sind, um eine virtuelle Umgebung zu erschaffen, eingegangen. Dies sind vor allem die Eingabegeräte zur Interaktion mit der künstlichen Welt und die Ausgabegeräte zur Immersion in eine solche Welt. Desweiteren werden unterschiedliche Arten von VR-Systemen erläutert. Diese reichen von simplen Desktop-Anwendungen bis hin zu aufwendigen Systemen bei denen ein fast vollständiges Eintauchen in eine virtuelle Umgebung ermöglicht wird. Das Kapitel wird abgeschlossen mit der Beschreibung einiger Einsatzmöglichkeiten für diese Technologien.

Im Kapitel *Persistenz* werden neben einer dazugehörigen Begriffserklärung unterschiedliche Ansätze zur Umsetzung eben jener Persistenz beschrieben. Die aufgezeigten Verfahren unterscheiden sich mitunter enorm.

Anschließend folgt das Kapitel *Virtual Reality Modeling Language (VRML)*. Es beschäftigt sich mit dieser abstrakten Beschreibungssprache, da VRML teilweise auch als eine Art Austauschformat für 3D-Daten verwendet wird. Neben der Entstehungsgeschichte von VRML und dessen Weiterentwicklung, erfolgt auch eine kleine Einführung in die Funktionsweise dieser Sprache und eine Betrachtung der vorhandenen Persistenzmöglichkeiten. Abschließend wird in diesem Kapitel die Verwendbarkeit von VRML, im Bezug auf die Aufgabenstellung dieser Diplomarbeit, überprüft.

Da ein eigenständiger XML-Dialekt als Datenformat für die Szeneninformationen alle Anforderungen erfüllt und gleichzeitig viele Vorteile bietet, befasst sich das nächste Kapitel namens *eXtensible Markup Language (XML)* mit genau dieser Auszeichnungssprache. Es beinhaltet, neben dem grundlegenden Konzept von Auszeichnungssprachen, eine Einführung in XML und XSL. Die Vorteile dieser Technologie werden gegen Ende des Kapitels noch einmal aufgeführt.

Das Kapitel *Die Software-Schnittstelle* befasst sich schließlich mit dem praktischen Teil dieser Diplomarbeit. Neben einer kurzen Beschreibung von Avango, wird hier natürlich auf die entwickelte Klasse für die Serialisierungs- und Deserialisierungs-Funktionen eingegangen. Außerdem wird noch das XML-Schema, welches den XML-Dialekt definiert, kurz angesprochen.

Zur Umsetzung der Projektanforderungen wurde die vorhandene VR-Entwicklungsumgebung Avango nur minimal verändert. Besonders Kapitel 3 und Kapitel 6 zeigen deutlich, was bei der Realisierung eines Persistenzkonzepts zu beachten ist. Auch die verwendete XML-Technologie gewährleistet eine hohe Kompatibilität zu externen Anwendungen bzw. Formaten. Wodurch wiederum keine neuen Anpassungen nötig werden, auf der anderen Seite aber auch außenstehenden Benutzern ein vereinfachter Zugang in die Anwendungsentwicklung mit Avango ermöglicht wird.

7.2 Ausblick

Durch die, vermutlich auch in Zukunft, stetig steigende Nachfrage nach immer leistungsfähigeren und realistischeren VR-Anwendungen und die gleichzeitige Entwicklung neuer Technologien, werden die Anforderungen an die Entwickler von VR-Anwendungen immer höher. Es werden fähige Programmierer benötigt, um die Möglichkeiten der neuesten Technologien völlig ausschöpfen zu können. Auf der anderen Seite bedarf es Künstler, um eine virtuelle Szene so realistisch wie nur möglich, zu modellieren. Nun gibt es allerdings wenige Menschen, die beide Talente bzw. Fähigkeiten in sich vereinen. Auch daran wird sich vermutlich nichts ändern in Zukunft.

Aus diesem Grund bedarf es Mechanismen, die die Entwicklung von VR-Anwendungen vereinfachen und somit für eine größere Personengruppe möglich machen. In den meisten Fällen wird die anwendungsspezifische komplexe Datenstruktur eines Szenengraphen durch Bearbeitungswerkzeuge vereinfacht repräsentiert um leichter verständlich für den Menschen zu sein. Diese vereinfachte Darstellung kann dann beliebig manipuliert und anschließend gespeichert werden. Um nun aber überhaupt virtuelle Szenen speichern und laden zu können, benötigt ein VR-System eine Persistenzschnittstelle. Andernfalls wäre ein Zugriff auf die Datenstruktur nur innerhalb der laufenden Anwendung möglich und da auch nur mit den Möglichkeiten eines komplexen VR-Systems, was wiederum einige Programmierkenntnisse voraussetzt.

Somit ist und bleibt eine Persistenz-Schnittstelle auch außerhalb des ProVíT - Projekts eine nützliche und wichtige Software-Komponente. Auch die Vereinfachung des Entwicklungsprozesses wird in Anbetracht von möglichen Kosteneinsparungen durch kürzere Entwicklungszeiten weiterhin vorangetrieben. Aber auch schon alleine wegen der ständig neuen Technologien sind Vereinfachungsmechanismen unabdingbar.

Bezüglich der Implementierung im Rahmen dieser Diplomarbeit bleibt zu erwähnen, das in Zukunft die Konfigurationsmöglichkeiten des Xerces-Parser besser ausgenutzt werden könnten, bzw. überhaupt die Möglichkeit dazu integriert wird. Außerdem könnte der rekursive Lösungsansatz in Anbetracht der Performanz durch eine iterative Lösung ersetzt werden. Beispielsweise beim Einlesen von XML-Daten durch zweimaliges Durchlaufen des Dokuments. Zunächst um alle nötigen Objekte zu erzeugen

und ein zweites Mal um die anfallenden Verbindungen zwischen den Objekten zu rekonstruieren.

Das XML-Schema ließe sich auch erweitern. Sinnvollerweise im Bezug auf eine konkretere Definition der unterschiedlichen Datentypen mitsamt eines gültigen Wertebereiches. Unter anderem könnte man auch das Element *fpFieldContainer* im XML-Schema durch den genauen Objekttyp ersetzen. Dies wäre eine minimale Änderung innerhalb der Software und würde die eindeutige Definition eines jeden FieldContainers und seiner Felder innerhalb des Schemas erlauben bzw. sogar notwendig machen.

7.2.1 XSL-Transformationen

Um nun die gespeicherten XML-Daten auch anderweitig Nutzen zu können, können sie mit XSLT umgewandelt werden. Alles was dazu nötig ist, ist ein XSLT-Prozessor. Dieser kann aus einer XML-Datei und einem XSL-Stylesheet, wie in Kapitel 5 bereits angesprochen, ein neues Ausgabedokument erzeugen. Es gibt verschiedene XSLT-Prozessoren, einige davon sind:

- FOP[18], dieser XSLT-Prozessor von Apache kann XML-Daten in folgende Formate umwandeln: PDF, PCL, PS, SVG, XML, AWT, MIF und TXT. [FOP04]
- MSXSL[19], ein XSLT-Prozessor von Microsoft
- Saxon[20]
- Xalan[21], auch von Apache, kann XML-Daten in HTML, Text oder andere XML-Dialekte umwandeln.

Bei der Transformation wird grob nach folgendem Verfahren vorgegangen. Der Quellbaum mit den XML-Daten wird rekursiv von der Wurzel beginnend verarbeitet. Beim durchschreiten der Baumstruktur wird dann für jeden angetroffenen Knoten eine zuvor

[18] http://xml.apache.org/fop/
[19] http://msdn.microsoft.com/xml
[20] http://saxon.sourceforge.net/
[21] http://xml.apache.org/xalan-c/index.html

festgelegte Regel angewendet. [Kre03] Existiert eine solche Regel für einen Knoten nicht, so wird dieser Knoten in der Umwandlung ignoriert. Damit ließen sich z. B. überflüssige Informationen für eine bestimmte Repräsentation der Daten oder für ein bestimmtes Zielformat herausfiltern.

Eine einfache Regel sieht wie folgt aus:

```
<xsl:template match="fpFieldContainer">
  ... Anweisungen ...
</xsl:template>
```

Diese Regel würde bei allen *fpFieldContainer-Elementen* die festgelegten Anweisungen ausführen. Als Auswahlparameter können entweder einfach Elementnamen angegeben werden, oder aber auch Attribute. Letztere allerdings mit einem „@“ vor dem Attributnamen. Es gibt allerdings noch viele verschiedene Möglichkeiten um komplexe Regeln aufzustellen.

Als einfaches Beispiel hier nun mal eine Umwandlung in HTML zur Veranschaulichung von XSLT. HTML steht zwar in keinerlei Zusammenhang zu einem brauchbaren 3D-Datenformat, allerdings ist das Prinzip einer solchen Umwandlung stets gleich und somit unabhängig vom Ausgabeformat.

```
<xsl:template match="/">
  <html>
    <head>
    </head>
    <body>
      <table style="font-family:Verdana; font-size:8pt; color:880000">
        <xsl:apply-templates />
      </table>
    </body>
  </html>
</xsl:template>
```

“/” bezeichnet die Wurzel des Quelldokuments und führt in diesem Fall dazu, das die Struktur einer HTML-Datei aufgebaut wird. Die Anweisung *<xsl:apply-templates />* sorgt dafür, das eine Liste der Kindknoten des aktuellen Knotens verarbeitet wird.

```
<xsl:template match="fpFieldContainer">
  <tr style="font-weight:bold">
    <th colspan="2">
      <xsl:value-of select='@type' /> (ID: <xsl:value-of
        select='@id' />):
      <xsl:apply-templates />
    </th>
  </tr>
</xsl:template>
```

Für jeden *fpFieldContainer* werden die Attribute *type* und *id* ausgewählt. Dies geschieht z. B. durch *<xsl:value-of select='@type' />*. Anschließend werden diese Informationen ausgegeben. Auch innerhalb dieser Regel wird die Verarbeitung der Kindknoten gestartet.

```
<xsl:template match="fpField">
  <tr style="font-weight:normal">
    <td><xsl:value-of select='@name' /></td>
    <td><xsl:value-of select='.' /></td>
  </tr>
</xsl:template>
```

Bei *fpField-Elementen* wird schließlich deren Attribut *name* und ihr Wert durch „.“ selektiert und ausgegeben.

Auf diese Art und Weise lassen sich also XML-Daten umwandeln. Natürlich sieht das im Regelfall deutlich komplexer aus als hier, aber die Vorgehensweise bleibt letztendlich identisch. Somit sind die Daten theoretisch kompatibel zu jedem Format. Außerdem werden dadurch auch externe Anwendungen in die Lage versetzt, lediglich die für sie relevanten Daten aus einer größeren Datenmenge herauszufiltern. Sei dies nun um die Szene einfach nur in einer gröberen Granularität für den Menschen darzustellen oder aber auch im Rahmen eines Modellierungswerkzeugs.

Literaturverzeichnis

[ABC83] Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. & Morrison, R. *An Approach to Persistent Programming* Computer Journal 26, 4 1983

[Apa04] Homepage: The Apache Software Foundation *Xerces C++ Parser* Juli 2004 URL: http://xml.apache.org/xerces-c/index.html

[Bei04] Klaus-Peter Beier *Virtual Reality: A Short Introduction* Februar 2004 URL: http://www-vrl.umich.edu/intro/

[BO00] Philipp von Bassewitz, Martin Obermeier *Document Object Model* Technische Universität München November 2000 URL: http://wwweickel.in.tum.de/lehre/Seminare/Hauptseminar/WS00/DOM/Presentation/paper.html#toc-2

[Bos98] Jon Bosak *Media-Independent Publishing: Four Myths about XML* Oktober 1998 URL: http://www.ibiblio.org/pub/sun-info/standards/xml/why/4myths.htm

[BT99] Jon Bosak, Tim Bray *XML and the Second-Generation Web* Mai 1999 URL: http://www.sciam.com/print_version.cfm?articleID=0008C786-91DB-1CD6-B4A8809EC588EEDF

[Cav00] Damien Cave *Artificial Stupidity* Oktober 2000 URL: http://dir.salon.com/tech/feature/2000/10/04/lanier/index.html

[CNG02] Carolina Cruz-Neira, Timothy Griepp *XJL - an XML Schema for the Rapid Development of Advanced Synthetic Environments* 2002

[CO94] Stephen Crawley, Michael Oudshoorn *Orthogonal Persistence and Ada* University of Adelaide Juli 1994

[CO95] Stephen Crawley, Michael Oudshoorn *Persistence Extensions to Ada* University of Adelaide Juli 1995

[Deb03] Uwe Debacher *VRML-Einführung* 2003
URL: http://www.debacher.de/vrml/vrml.htm

[DLR04] Homepage: DLR - Institute of Robotics and Mechatronics *The DLR SpaceMouse* Mai 2004 URL: http://www.robotic.dlr.de/mmi/sm/

[Eib01] Dr. Maximilian Eibl *X3D-Standard: 3D im Internet* 2001
URL: http://www.tecchannel.de/internet/760/

[Fau97] Martin Faust *Virtuelle Welten* Linux Magazin Dezember 1997 URL: http://www.linux-magazin.de/Artikel/ausgabe/1997/12/VLE/vle.html

[FOP04] Homepage: The Apache XML Project *FOP* August 2004
URL: http://xml.apache.org/fop/

[FSL04] Homepage: Fakespace Labs *Research Tools Created* Mai 2004
URL: http://www.fakespacelabs.com/tools.htm

[Gal03] Goran Galunic *Office of the Future - Einbettung virtueller Hilfsmittel in Arbeitsumgebungen am Beispiel der Entwicklung einer Visualisierungs- und Kommunikationsschnittstelle für ein Mixed Reality Conferencing System* Fachhochschule Köln, Abteilung Gummersbach August 2003

[HW96] Jed Hartman, Josie Wernecke *The VRML 2.0 Handbook - Building Moving Worlds on the Web* Addison-Wesley 1996

[Idr99] Nazmul Idris *Should I use SAX or DOM?* Mai 1999
URL: http://www.developerlife.com/saxvsdom/default.htm#24983

[IM04] Homepage: Ikarus Mediothek *Einführung in die Thematik von Virtual Reality* Mai 2004 URL: http://www.ikarus.uni-dortmund.de/Archiv/Virtual%20Reality/Online-Seminare/Einfuehrung%20in%20die%20Thematik/Einfuehrung%20in%20die%20Thematik.shtml

[IMK04] Homepage: Fraunhofer IMK *i-CONE - Displaysystem für virtuelle Umgebungen* Mai 2004 URL: http://www.imk.fraunhofer.de/sixcms/detail.php?template=&id=1336#top

[Kel04] Wolfgang Keller *Persistence Options for Object-Oriented Programs* 2004

[Kre03] Thomas Kretschmer *Wahlpflichtvorlesung XML* 2003

[KRS98] Jörg H. Kloss, Robert Rockwell, Kornél Szabo, Martin Duchrow *VRML 97 - Der neue Standard für interaktive 3D-Welten im World Wide Web* Addison-Wesley, München 1998

[Lea99] Anne C. Lear *XML Seen as Integral to Application Integration* IT Pro, S. 12ff. September/Oktober 1999

[LS03] Melanie O. Läge, Attila S. Suicmez *Skript Virtuelle Realität* April 2003 URL: http://userpage.chemie.fu-berlin.de/~sunny/svr/default.html

[Mai99] Klaus Mainzer *Computernetze und virtuelle Realität - Leben in der Wissensgesellschaft* Springer Verlag 1999

[MC99] Paul Milgram, Herman Colquhoun Jr. *A Taxonomy of Real and Virtual World Display Integration* University of Toronto, 1999

[MCC94] Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Munro, D. *Delivering the Benefits of Persistence to System*

Construction and Execution In Proc. 17th Australasian Computer Science Conference, Christchurch, New Zealand 1994

[NL03] Homepage: Net-Lexikon.de *Virtuelle Realität - Definition, Bedeutung, Erklärung im Lexikon* Oktober 2003
URL: http://www.net-lexikon.de/Virtuelle-Realitaet.html

[Pes97] Mark Pesce *VRML - Cyberspace-Welten erkunden und erschaffen* Hanser Fachbuch Februar 1997

[PHP04] Homepage: PHP.net *PHP: Hypertext Preprocessor* Juli 2004
URL: http://www.php.net/

[Pro04] Homepage: ProViT *Kurzeinführung zu ProViT* Juli 2004
URL: http://www.provit.net/provit_intro_deutsch.pdf

[Rhe95] Howard Rheingold *Virtuelle Welten – Reisen im Cyberspace* Rowohlt Taschenbuch Verlag GmbH, Reinbeck bei Hamburg März 1995

[Roc99] Robert Rockwell *VRML: Beyond the Desktop* 1999
URL: http://www.aquileia.it/webtec.htm

[SGI04] Homepage: Silicon Graphics *OpenGL Performer - Frequently Asked Questions* August 2004
URL: http://www.sgi.com/products/software/performer/faq.html

[SH01] Homepage: SelfHTML *XML/DTDs* 2001
URL: http://de.selfhtml.org/xml/

[SPI04] Homepage: SPI Systemberatung *Hardware: SpaceMouse* Mai 2004 URL: http://www.spi.de/hardware/spacemouse.htm

[ST98] Homepage: STREETtech.com *VPL Research, Inc.* 1998
URL: http://www.streettech.com/bcp/BCPgraf/StreetTech/VPL.html

[SW00] Erwin Schuster, Stephan Wilhelm *Content Management*
Informatik Spektrum, Springer-Verlag Berlin Heidelberg 2000

[Tra99] Henrik Tramberend *Avango: A Distributed Virtual Reality Framework*
März 1999

[UIC04] Homepage: The Mississippi Center for Supercomputing Research
UUID(3C) August 2004
URL: http://www.mcsr.olemiss.edu/cgi-bin/man-cgi?uuid+3

[UIG04] Homepage: Die.net *uuid_generate(3) - Linux man page* August 2004
URL: http://www.die.net/doc/linux/man/man3/uuid_generate.3.html

[Uni04] Homepage: Unicode.org *Unicode Home Page* August 2004
URL: http://www.unicode.org

[US04] Homepage: Universität Stuttgart, Institut für Computeranwendungen
VRML Juli 2004 URL: http://www.csv.ica.uni-stuttgart.de

[VT04] Homepage: Vapour Technology Ltd. *PHP/VRML Workshop*
2001 URL: http://web3d.vapourtech.com/workshop/php-vrml/

[W3S04] Homepage: W3Schools *W3Schools Online Web Tutorials*
August 2004 URL: http://www.w3schools.com/